Nr 1/2005 (30) marzec

Biblioteczka

Komputer Świat

NR INDEKSU: 321958 ISBN 83-89371-33-2

tylko 11,80 zł w tym 0% VAT

Z płytą CD-ROM

Szkoła pisania programów od podstaw

Programowanie w C++

Na płycie aplikacje niezbędne do pisania programów w C++ oraz do wykonania wskazówek z książki

- Co powinien wiedzieć programista
- Najważniejszy jest dobry plan
- Instrukcje, funkcje, tablice i wskaźniki
- Podstawy projektowania obiektowego
- Słowniczek terminów programistycznych

Biblioteczka

Programowanie w C++

Na płycie środowiska programistyczne języka C++

Wymagania sprzętowe

- Komputer PC z procesorem Intel Pentium lub kompatybilnym
- 64 MB lub więcej pamięci operacyjnej RAM
- Minimum 250 MB wolnego miejsca na dysku twardym
- Napęd CD-ROM
- Monitor i karta graficzna VGA lub lepsza
- Klawiatura i mysz (ewentualnie inne urządzenie wskazujące)
- Zalecany dostęp do internetu

Biblioteczka Komputer Świat

Programowanie w C++

Szkoła pisania programów od podstaw

Spis treści

Wstęp 4

Od autora 4

1 Czym jest programowanie 6

Jak dogadać się z komputerem 7
Programować w C++ 8
Kompilacja programu 10
Przygotowujemy komputer do programowania 12
Dobrze wszystko zaplanuj 16

2 Szkielet programu 20

Pierwszy program 21
Dyrektywa include 26
Funkcja main () 27
Znaki do zadań specjalnych 28
Kod z komentarzem 30

3 Podstawy języka C++ 32

Wyświetlone na ekranie 33
Stałe i zmienne 34
Wczytywanie z klawiatury 37
Operacje na zmiennych 39
Nasz pierwszy przydatny program 46
Typy danych 48

4 Sterowanie programem 54

Instrukcje warunkowe 55
Zapętlony program 66

5 Funkcje 82

Co to jest funkcja 83
Widoczność zmiennych 92
Przydatne funkcje matematyczne 95

6 Tablice, wskaźniki i inne typy 96

Niezwykłe tablice 97
Wskaźniki 104
Przekazywanie wskaźników do funkcji 109

7 Podstawy programowania obiektowego .. 112

Programowanie obiektowe 113
Tworzymy aplikację 114
Ustawienia C++ Buildera 120

Dodatek 122

Błędy w kodzie 122

Leksykon 126

Paweł Paczuski

Od autora

Programowanie to niezwykle fascynująca dziedzina informatyki. Łączy w sobie drzemiące w każdym człowieku potrzeby: tworzenia czegoś nowego i upraszczania sobie życia. Ale czy każdy z nas może zostać programistą? Oczywiście, że tak. Może nie od razu napiszemy system operacyjny, ale stworzenie niewielkiej, pożytecznej aplikacji nie powinno nam sprawić większych problemów. Pod warunkiem, że poznamy podstawy języka programowania.

Wiedza zawarta w tej książce została przekazana w bardzo przystępny sposób, znany wielu Czytelnikom z poprzednich publikacji Komputer ŚWIATA. Książka została podzielona na siedem rozdziałów, w których krok po kroku poznamy kolejne, coraz bardziej złożone zagadnienia związane z programowaniem w języku C++. Nie jest to jednak czysta teoria, ale całkowicie praktyczna wiedza zilustrowana kilkudziesięcioma przykładami.

Dla kogo przeznaczona jest ta książka? Dla wszystkich, którzy chcą tworzyć własne programy, ale nie wiedzą, od czego zacząć. Sięgnąć po nią powinni także uczniowie i studenci dostający gęsiej skórki na samą myśl o konieczności zaliczenia programowania – przedmiotu, który w ostatnich latach ze studenckich audytoriów przywędrował także do liceów, a nawet wielu gimnazjów.

Mam nadzieję, że książka zachęci młodych adeptów sztuki programowania do poszerzania swojej wiedzy. Może po jej lekturze zainteresowanie językiem C++ przerodzi się w fascynację i zaowocuje powstaniem ciekawej i pożytecznej aplikacji, z której korzystać będą miliony użytkowników komputerów.

1 Czym jest programowanie

Dowiedzmy się, od czego zacząć naszą przygodę z programowaniem

W rozdziale	strona
Jak dogadać się z komputerem	**7**
Język programowania	7
Programować w C++	**8**
Historia C++	8
Zalety języka C++	9
Blok, struktura i obiekt	9
C++ w standardzie	10
Kompilacja programu	**10**
Przygotowujemy komputer do programowania	**12**
Instalacja Dev-C++	12
Poznajemy Dev-C++	14
Dobrze wszystko zaplanuj	**16**
Jak tworzyć schematy blokowe	18

Wiele osób uważa, że programowanie to tak skomplikowane zajęcie, że nie warto tracić czasu na jego naukę. Komputer ŚWIAT postara się udowodnić, że przy odrobinie dobrej woli programowaniem może zająć się każdy.

Pisanie programów to wcale nie czarna magia, tylko bardzo logiczna rozmowa z komputerem. Oczywiście w jednym ze specjalnie stworzonych do tego celu języków. Dlatego pierwszym krokiem jest wybór języka programowania. Kolejne zadanie, które czeka przyszłego programistę, to instalacja aplikacji, która ułatwi nam tworzenie kodu programu, a następnie skompiluje i stworzy plik, który będziemy mogli uruchomić. Komputer ŚWIAT podpowiada, w jaki sposób wybrać zarówno język, jak i odpowiednią aplikację wspomagającą pisanie kodu.

Na końcu dowiemy się również, od czego zacząć pisanie programu – nauczymy się między innymi tworzyć schematy blokowe naszych programów.

Jak dogadać się z komputerem

Komputer nie myśli. On tylko wykonuje nasze polecenia. Nauczmy się więc wydawać mu rozkazy

Terminem programowanie można określić wszystko, co przyczynia się do powstania gotowej aplikacji. Programowanie polega więc na zmuszeniu komputera do wykonywania określonych przez nas poleceń. Jak to zrobić? Wystarczy przekazać mu nasze polecenia w języku, który będzie dla niego zrozumiały.

Język programowania

Komputer to maszyna niezwykle logiczna – wykonuje wyłącznie ściśle określone czynności. Ale w jaki sposób poinformować go o tym, co powinien zrobić? Przecież komputer nie zrozumie polecenia wydanego mu po angielsku, a tym bardziej po polsku. Prawdę mówiąc, procesor, który jest mózgiem komputera, jest w stanie zrozumieć tylko polecenia zapisane w systemie dwójkowym, czyli w postaci zer i jedynek. Nie martwy

Od języka ludzkiego do kodu maszyny

poziom wysoki

poziom niski

Język ludzki (mowa)

```
Jeśli liczba punktów jest
większa niż 50, test zostaje
zaliczony. W przeciwnym
wypadku brak zaliczenia
```

Język programowania wysokiego poziomu

```
if (liczba_punktów>50)
  zaliczenie_testu=true
else
  zaliczenie_testu=false
```

Język programowania niskiego poziomu

```
mov ecx, 50
dec ecx
push ecx
pop eax
or ecx, eax
```

Język maszynowy (kod binarny)

```
01001101001110101011 0
10111011101110111011 1
00111011101010111011 00
11101010101110010100 10
10110100101000101001 00
```

się jednak, że będziemy musieli nauczyć się posługiwania ciągiem zerojedynkowym. Dzięki systemowi operacyjnemu i istniejącym już aplikacjom wspomagającym programowanie podczas przekazywania komputerowi poleceń posłużymy się językiem zbliżonym bardziej do ludzkiej mowy niż ciągów składających się z zer i jedynek. Taki język nazywa się językiem wysokiego poziomu (patrz schemat na poprzedniej stronie). Warto wiedzieć, że istnieją również języki niskiego poziomu, w których jedno polecenie odpowiada jednemu rozkazowi procesora. Przykładem takiego języka jest Asembler – powstał na bazie języka maszynowego, przez zastąpienie ciągów zer i jedynek bardziej zrozumiałymi poleceniami. Programowanie z wykorzystaniem języka niskiego poziomu jest bardzo trudne, ponieważ wymaga szczegółowej znajomości budowy procesora i pozostałych elementów komputera.

System dwójkowy

Komputer zbudowany jest podobnie jak człowiek. Jego mózg (procesor) steruje wszystkimi pozostałymi elementami. Informacje przekazywane z procesora do innych komponentów mają postać impulsów elektrycznych. Tak więc cały język wewnętrzny komputera składa się z dwóch informacji – jest sygnał lub go nie ma. Do reprezentacji takiego języka wystarczą dwie cyfry: 1 i 0. Taki system komunikacji nazywa się dwójkowym i można za jego pomocą przekazać dowolną informację. Liczbę dwójkową można zamienić na liczbę dziesiętną, dodając do siebie kolejne potęgi liczby 2. Na przykład, dwójkowej liczbie 10111 odpowiada dziesiętna liczba 23, ponieważ:

$10111 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 16 + 0 + 4 + 2 + 1 = 23$

Programować w C++

Przed przystąpieniem do programowania musimy wybrać język, którym będziemy się posługiwać. Zobaczmy, jak to zrobić

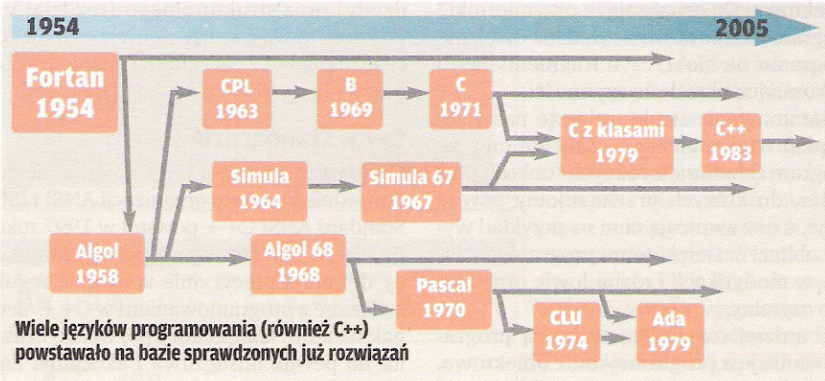
Każdy programista musi wybrać język programowania, za pomocą którego będzie się porozumiewał z komputerem. Jak już ustaliliśmy, będzie on należał do rodziny języków wysokiego poziomu. Wybór jednak nie jest łatwy – istnieje bowiem kilkaset różnych języków, a ich odmian i dialektów nie sposób zliczyć. W rzeczywistości wybór sprowadza się jednak do kilku pozycji. A ponieważ najpopularniejszy jest język C++, dlatego właśnie od niego najlepiej zacząć naukę programowania.

Historia C++

Zanim zaczniemy naukę, poznajmy historię powstania języka C++. Na jej podstawie przekonamy się, że nie jest on tworem wymyślonym jednego dnia przez szalonego informatyka, lecz ulepszoną i dopracowaną wersją wcześniej istniejących języków programowania.

Pomysłodawcą i projektantem języka C++ jest Bjarne Stroustrup (czytaj bjarne strowstrup). Pierwsza wersja C++ (znana pod nazwą C z klasami) pojawiła się w roku 1979. Została zaprojektowana na bazie dwóch innych języków – Simuli 67 i C. Jednak na jej rozwój miały

Historia języków programowania



Wiele języków programowania (również C++) powstawało na bazie sprawdzonych już rozwiązań

wpływ również inne języki: Algol 68, Ada, ML czy CLU.

Nazwa C++ została po raz pierwszy użyta w 1983 roku przez Ricka Mascittiego i jest niezwykle trafna. Jak się niedługo przekonamy, dwa plusy oznaczają zwiększenie o jeden, a więc zapis C++ oznacza po prostu następną wersję języka C.

Zalety języka C++

Skoro ustaliliśmy już, że będziemy programować w języku C++, warto poznać jego zalety:

- czytelność – ponieważ język C++ jest zbliżony do języka ludzkiego, kod programu jest łatwy do odczytania,
- przenośność – napisane przez nas programy bez trudu uruchomimy na komputerach różnych typów, które pracują pod różnymi systemami operacyjnymi,
- popularność – w języku C++ powstaje większość oprogramowania dostępnego na rynku, a poznanie zasad rządzących programowaniem w C++ znacznie ułatwia naukę innych języków (patrz ramka Podobieństwa między językami).

Blok, struktura i obiekt

Jeśli przeglądaliśmy książki dotyczące programowania, z pewnością napotkaliśmy kilkakrotnie różne przymiotniki określają-

Podobieństwa między językami

Ustaliliśmy, że będziemy się uczyć programować w języku C++. To wcale nie oznacza, że opanowanie innego języka programowania będzie wymagało jego nauki od podstaw. Popularne języki programowania, takie jak C++, Pascal, Java, Basic czy PHP, są bowiem bardzo do siebie podobne. Oczywiście często różnią się podejściem do programowania i szczegółami dotyczącymi budowy różnych konstrukcji językowych. Jednak niemal wszystkie omawiane w tej książce zasady budowania programów blokowych i strukturalnych w C++ można z powodzeniem stosować w pozostałych językach.

ce technikę programowania – na przykład programowanie blokowe, strukturalne czy obiektowe. Co oznaczają te przymiotniki? Programowanie blokowe zakłada dzielenie programu na bloki, czyli fragmenty kodu wykonujące określone czynności.

Programowanie strukturalne to następny stopień wtajemniczenia. Zakłada ono, że program składa się z różnych funkcji i procedur, do których przekazujemy pewne dane, a one zwracają nam na przykład wynik obliczeń. Dzięki temu program jest łatwy w modyfikacji i rozbudowie oraz bardzo czytelny.

Najbardziej rozwiniętą techniką programowania jest programowanie obiektowe. Zakłada ono, że program składa się z obiektów, które mają pewne własności i mogą wykonywać określone czynności.

W tej książce poznamy tajniki programowania blokowego (pierwsze cztery rozdziały) oraz strukturalnego (rozdział V). Ze zdobytą wiedzą będziemy mogli rozpocząć przygodę z obiektami (przedsmak tej przygody znajdziemy w rozdziale VII).

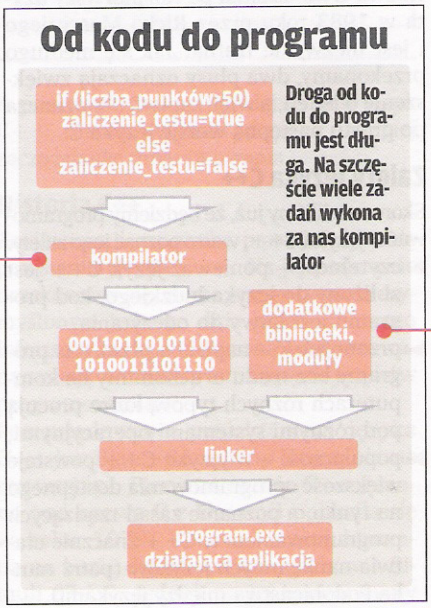
C++ w standardzie

Standaryzacją języka C++ zajmują się odpowiednie komitety organizacji ANSI i ISO. Standard ANSI C++ powstał w 1997 roku, ISO C++ narodziło się rok później. Standardy definiują precyzyjnie wszystkie reguły związane z programowaniem w C++. Jednak niektóre kompilatory pozwalają czasami na pewne odstępstwa i akceptują kod niezgodny ze standardem. Komputer ŚWIAT będzie starał się przekazywać wiedzę o języku C++ zgodnie ze standardem ISO.

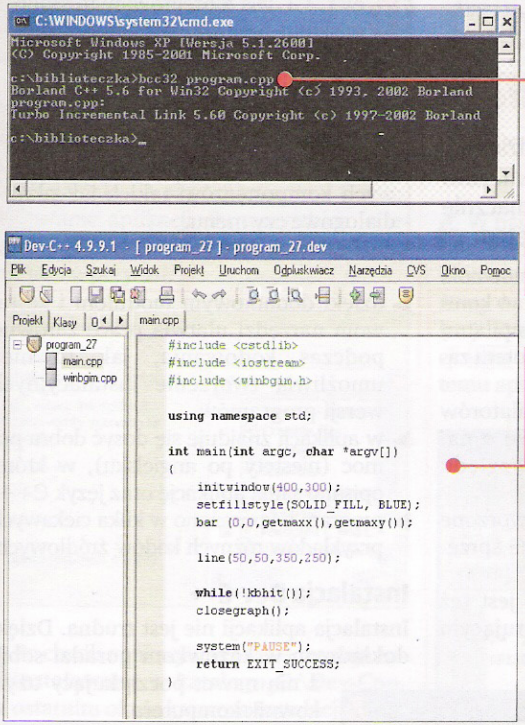
Kompilacja programu

Komputer nie rozumie języka C++. Musimy go więc przetłumaczyć na język zrozumiały dla maszyny

Program komputerowy to dla komputera ciąg poleceń. Polecenia te wyrażamy w określonym języku programowania (w naszym wypadku będzie to język C++) i nazywamy kodem źródłowym naszego programu. Jednak język programowania wysokiego poziomu nie jest jasny dla komputera, który rozumie jedynie ciągi zer i jedynek. Dlatego musimy posłużyć się aplikacją, która przetłumaczy zapisany przez nas kod na język niskiego poziomu, a w końcu na kod zrozumiały dla maszyny. Taką aplikacją jest kompilator. Jednak kompilator nie jest w stanie



Korzystamy z różnych kompilatorów języka C++



Na rynku dostępnych jest wiele różnych kompilatorów języka C++. Odpowiednie aplikacje znajdziemy dla każdego systemu operacyjnego – MS-DOS-u, Windows czy Linuksa. Wiele kompilatorów jest darmowych. Niektóre są bardzo proste, a korzystanie z nich wymaga wydania prostej komendy (na przykład ●). Wiele kompilatorów zostało wyposażonych w edytor kodu źródłowego ● (często potrafiący odpowiednio kolorować składnię kodu źródłowego, ułatwiając tym samym jego tworzenie i analizowanie). Istnieją też całe środowiska programistyczne, które często za cenę kilku tysięcy złotych oferują znaczną pomoc w procesie tworzenia aplikacji (w rozdziale VII poznamy możliwości i zalety darmowej wersji jednego z takich środowisk, która znajduje się również na płycie dołączonej do książki).

z kodu źródłowego stworzyć działającego programu. Do tego potrzebny jest jeszcze tak zwany linker. Jego zadaniem jest połączenie napisanego przez nas ciągu poleceń z dodatkowymi modułami i bibliotekami ●, które umożliwią współpracę naszej aplikacji z systemem operacyjnym. Dopiero po dołączeniu tych wszystkich dodatków zostanie stworzona wykonywalna wersja naszego programu – będziemy ją mogli w dowolnym momencie uruchomić i przetestować jej działanie. Proces kompilowania i łączenia może wydawać się skomplikowany, ale nie powinien nas odstraszyć od programowania. Kompilator i linker to w większości wypadków jedna aplikacja, którą uruchamia się pojedynczym poleceniem lub wciśnięciem określonego przycisku. Z następnych stron tej książki dowiemy się, który kompilator wybrać, jak go zainstalować oraz w jaki sposób z niego korzystać. Poznamy również wiele udogodnień stworzonych przez twórców kompilatorów, dzięki którym programowanie jest całkiem przyjemne.

Przygotowujemy komputer do programowania

Wiemy już, w jakim języku będziemy się porozumiewali z komputerem. Czas zainstalować narzędzie, które znacznie uprości dialog z naszym pecetem

Przed rozpoczęciem nauki programowania musimy odpowiednio przygotować komputer. Najpierw należy wybrać kompilator, z którego będziemy korzystali, a potem zainstalować go.

Spośród wielu dostępnych kompilatorów Komputer ŚWIAT poleca aplikację o nazwie Dev-C++. Dlaczego? Ze względu na kilka jej cech:

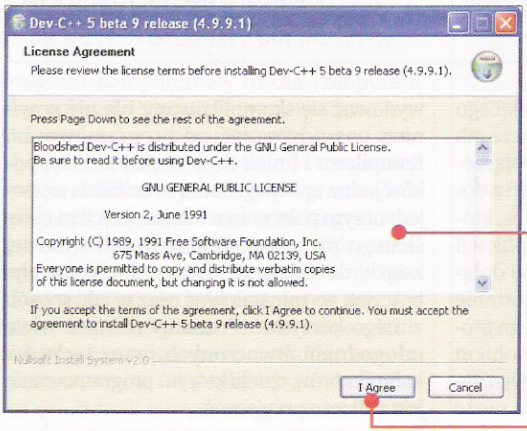
- jest całkowicie darmowa, a stworzone za jej pomocą programy można sprzedawać,
- oprócz zwykłego kompilatora jest też doskonałym edytorem kolorującym składnię kodu źródłowego,
- dzięki dodatkowym modułom i zestawom narzędzi ułatwia pracę zarówno podczas kodowania, jak również umożliwia tworzenie instalacyjnych wersji programów,
- w aplikacji znajduje się dosyć dobra pomoc (niestety po angielsku), w której opisano samą aplikację oraz język C++,
- program wyposażono w kilka ciekawych przykładów różnych kodów źródłowych.

Zintegrowane środowisko

Dev-C++ jest zintegrowanym środowiskiem programistycznym (IDE – z ang. Integrated Development Environment). Aplikacje będące IDE wyróżniają się tym, że udostępniają użytkownikowi możliwość edycji kodu źródłowego, kompilacji, a także tworzenia dodatkowych komponentów, takich jak okna dialogowe czy menu.

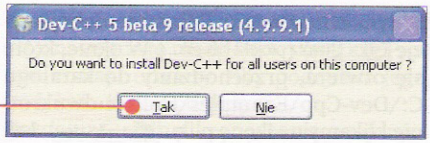
Instalacja Dev-C++

Instalacja aplikacji nie jest trudna. Dzięki dokładnym wskazówkom poradzi sobie z nią nawet początkujący użytkownik komputera.

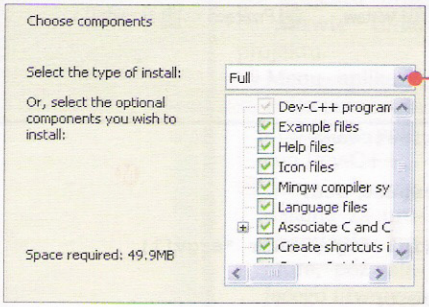


1 Wkładamy do napędu CD płytę dołączoną do tej książki. Następnie odnajdujemy i uruchamiamy plik o nazwie devcpp4991setup.

2 W oknie, które się pojawia, klikamy na przycisk OK. W następnym oknie możemy zapoznać się z warunkami licencji środowiska Dev-C++ ●. Aby przystąpić do instalacji programu, klikamy na przycisk ●.

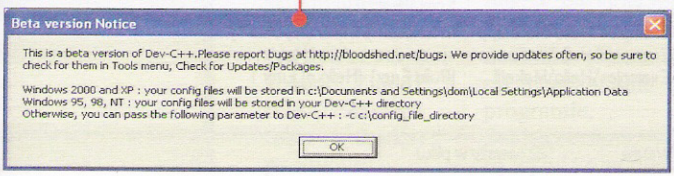


3 Z listy wybieramy pozycję Full (pełną instalację), klikamy na przycisk Next >, a w następnym oknie na przycisk Install. Jeśli pojawia się okno z pytaniem o zainstalowanie aplikacji dla wszystkich użytkowników naszego komputera, klikamy w nim na przycisk .



4 Po chwili aplikacja Dev-C++ zostaje zainstalowana w katalogu C:\Dev-Cpp. W ostatnim oknie kreatora instalacji klikamy na przycisk Finish – instalacja dobiega końca i Dev-C++ zostaje uruchomiony.

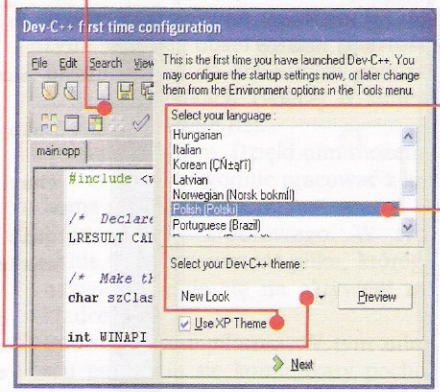
5 Przy pierwszym uruchomieniu aplikacji Dev-C++ należy ją skonfigurować. Na początku pojawia się okno , w którym klikamy na OK.



Komputer ŚWIAT radzi

Aby w przyszłości uruchomić aplikację, wystarczy z menu Start wybrać Programy (w Windows XP wybieramy Wszystkie programy), a następnie podmenu Bloodshed Dev-C++ i pozycję Dev-C++.

6 W następnym oknie możemy określić język aplikacji oraz jej wygląd. Z listy wybieramy pozycję Polish (Polski). Z listy możemy wybrać motyw (rodzaj wyglądu) aplikacji. Komputer ŚWIAT poleca wybór pozycji New Look i zaznaczenie opcji . Dzięki temu aplikacja będzie wyglądać tak .

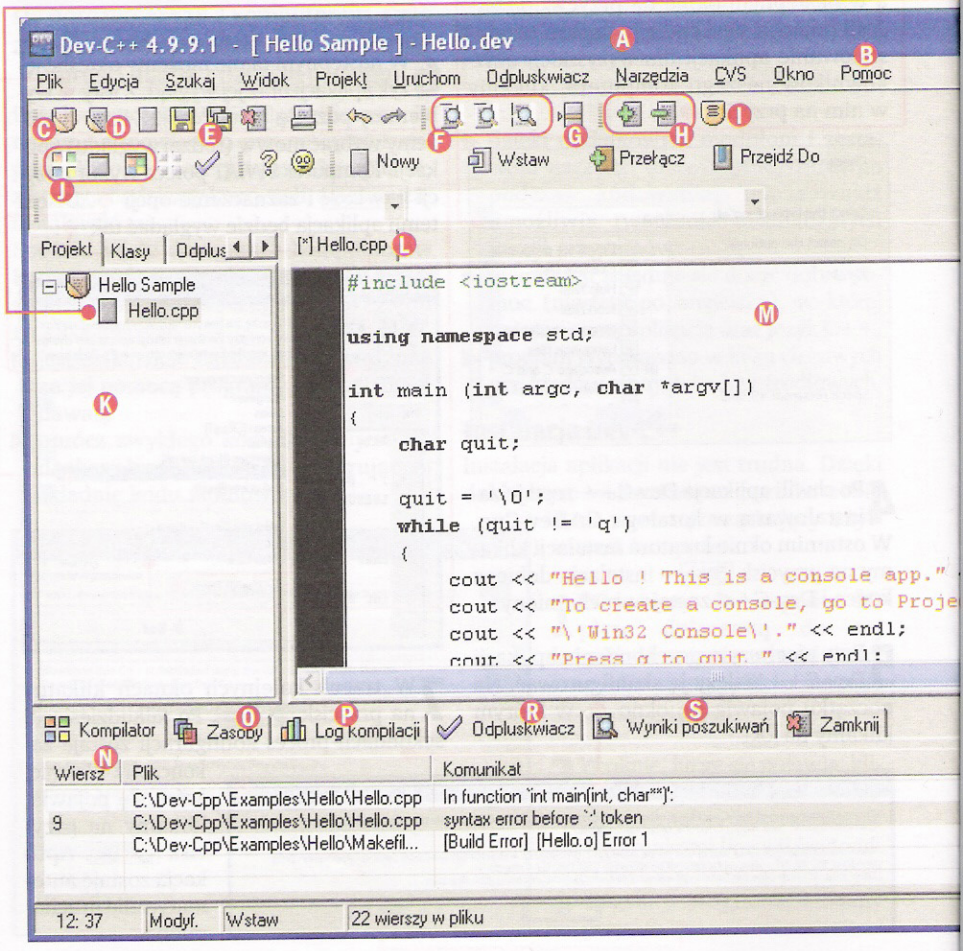


7 W trzech kolejnych oknach klikamy na przycisk Next. Po kilkudziesięciu sekundach proces konfiguracji zostaje zakończony. W oknie, które się pojawia, klikamy na przycisk OK. Aplikacja zostaje automatycznie uruchomiona.

Poznajemy Dev-C++

Nasz kompilator jest już zainstalowany i skonfigurowany. Dowiemy się teraz, jak go obsługiwać oraz poznamy znaczenie jego najważniejszych elementów i niektórych opcji.

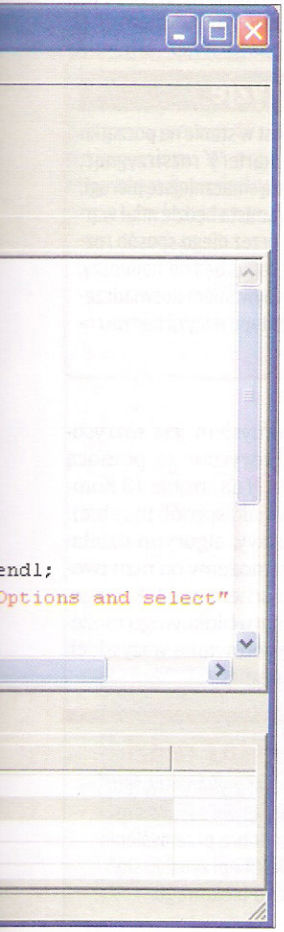
Aby na ekranie pojawiły się wszystkie ważne okna, otwieramy przykładowy projekt. W tym celu, z menu Plik wybieramy pozycję Otwórz projekt lub plik... W oknie, które się otwiera, przechodzimy do katalogu C:\Dev-Cpp\Examples\Hello i dwukrotnie klikamy na ikonę pliku Hello.dev. Jest to tak zwany plik projektu, w którym znajdują się informacje o nazwie pliku z kodem źródłowym oraz dodatkowe informacje



o projekcie (więcej o projekcie dowiemy się z następnego rozdziału tej książki). Projekt zostaje otwarty. Aby podejrzeć jego kod, klikamy na pozycję .

Poznajmy teraz przeznaczenie znajdujących się w aplikacji okien oraz znaczenie najważniejszych ikon.

A Belka tytułowa aplikacji. Oprócz informacji o wersji Dev-C++ znajduje się na niej również informacja o nazwie naszego projektu oraz o nazwie pliku projektu.



B Menu aplikacji, które umożliwia dostęp do poleceń Dev-C++.

C Ikona odpowiedzialna za uruchamianie kreatora, pozwalającego utworzyć nowy projekt.

D Ikona pozwalająca na otwieranie pliku lub projektu.

E Ikona umożliwiająca zapis wszystkich plików projektu.

F Ikony pozwalające na przeszukiwanie plików projektu (uruchamiają kolejno: Znajdź tekst, Zamień tekst i Szukaj ponownie).

G Ikona pozwalająca na przejście do dowolnego wiersza kodu źródłowego – z pewnością przyda się przy długich kodach.

H Ikony pozwalające na dodanie i usunięcie plików z projektu (więcej przeczytamy w rozdziale IV).

I Ikona opcji projektu.

J Ikony pozwalające na kompilację i uruchamienie stworzonego przez nas programu.

K Okno przeglądarki projektu. Znajdują się w niej nazwy wszystkich plików, z których składa się projekt. Po kliknięciu na zakładkę Klasy w oknie tym pojawiają się wszystkie stworzone w ramach projektu klasy, a po kliknięciu na zakładkę Odpluskwiacz pojawia się okno, w którym można obserwować wartości przypisywane do zmiennych podczas działania aplikacji (więcej o zmiennych w rozdziale III).

L Zakładki edytora. Dzięki nim możemy w Dev-C++ wygodnie pracować z kilkoma plikami naraz.

M Edytor kodu źródłowego. W tym oknie widać zawartość pliku, którego nazwa znajduje się na aktywnej zakładce L.

N Informacje kompilatora. W tym miejscu pojawiają się komunikaty pochodzące od kompilatora (na przykład komunikaty o błędach).

O Zakładka okna z informacjami dotyczącymi błędów kompilacji, które dotyczą plików zasobów projektu.

P Zakładka okna z logiem kompilatora.

R Zakładka okna z opcjami ułatwiającymi poprawianie błędów w naszym programie.

S Zakładka okna, w którym wyświetlane są wyniki wyszukiwania.

Dobrze wszystko zaplanuj

Wydając komputerowi rozkazy, trzeba być bardzo precyzyjnym. Warto więc dobrze zaplanować nasz kod

Programowanie to dość złożony proces. Aby się nie pogubić i jak najszybciej stworzyć poprawnie działającą aplikację, należy odpowiednio wszystko zaplanować. Aby lepiej zrozumieć proces planowania własnego programu, Komputer ŚWIAT proponuje kulinarne porównanie – brzmi to zabawnie, ale mniej zaawansowanym Czytelnikom powinno pomóc w zrozumieniu istoty planowania aplikacji.

1 Na początku musimy określić, co chcemy osiągnąć, czyli co nasz program ma wykonywać ❶. Jest to niezwykle ważna faza programowania i można ją porównać do wyboru posiłku, który mamy zamiar przygotować.

2 Jeśli wiemy już, co chcemy osiągnąć, warto sobie na kartce rozpisać, w jaki sposób program ma działać ❷. Powinniśmy stworzyć tak zwany algorytm, czyli dokładny opis tego, co program powinien po kolei robić, aby wykonywać określone przez nas zadanie. W kulinarnym porównaniu ustalamy, z czego składa się nasz posiłek, i odszukujemy odpowiednie przepisy. Ponieważ różnych sposobów na rozwiązanie określonego problemu jest co najmniej tyle, ile przepisów na przykład na pierogi, musimy wybrać ten według nas najlepszy.

▼ Trudne decyzje

Tak jak kucharz nie jest w stanie na początku swojej kulinarnej kariery rozstrzygnąć, z którego przepisu wyjdą smaczniejsze pierogi, tak początkujący programista będzie miał wątpliwości, czy wybrany przez niego sposób rozwiązania danego problemu będzie najlepszy. Na szczęście, wraz z nabywaniem doświadczenia podejmowanie tego typu decyzji będzie coraz łatwiejsze.

3 Bardzo dobrym pomysłem jest rozrysowanie naszego algorytmu za pomocą schematu blokowego ❸ (na stronie 18 Komputer ŚWIAT opisał, w jaki sposób to robić). Jeśli nabierzemy wprawy, algorytm działania naszego programu możemy od razu tworzyć właśnie w postaci schematu blokowego. Fazę tworzenia schematu blokowego możemy porównać do przygotowania wszystkich składników naszego dania.

► Komputer ŚWIAT radzi

Jeśli dopiero rozpoczynamy naszą przygodę z programowaniem, nie stawiajmy sobie zbyt wygórowanych celów. Lepiej na początku stworzyć program obliczający pole powierzchni jednej figury i go później rozbudowywać, niż od razu porywać się na aplikację rozwiązującą złożone układy równań i wykreślającą dla nich wykresy. Oczywiście, wraz z nabywaniem doświadczenia programistycznego, będziemy potrafili planować i kodować coraz bardziej rozbudowane aplikacje.

► Komputer ŚWIAT radzi

Im dokładniejszy schemat wykonamy, tym jego zakodowanie będzie łatwiejsze. Warto więc poświęcić czas na dobre przemyślenie algorytmu naszej aplikacji i przyłożyć się przy tworzeniu schematu blokowego.

► Komputer ŚWIAT radzi

Jeśli w napisanym przez nas kodzie źródłowym kompilator wykryje wiele błędów, nie zrażajmy się tym i poprawiajmy kolejne niedoskonałości kodu. Większość błędów to następstwa drobnych literówek, a tych trudno uniknąć nawet profesjonalistom.

4 Teraz możemy już przystąpić do kodowania, czyli przedstawienia schematu blokowego naszego programu za pomocą konkretnych poleceń wybranego języka programowania ❹. To główny etap pracy programisty. W porównaniu kulinarnym etap ten odpowiada gotowaniu czy smażeniu naszej potrawy.

5 Kolejnym etapem jest kompilacja i uruchomienie naszego programu (jeśli kompilacja przebiegła bez błędów) ❺. Kulinarnym odpowiednikiem tej fazy jest sprawdzenie, czy poszczególne składniki dania odpowiednio smakują, a następnie ułożenie ich na talerzu (oczywiście, jeśli coś było na przykład niesłone, należy dany składnik odpowiednio doprawić).

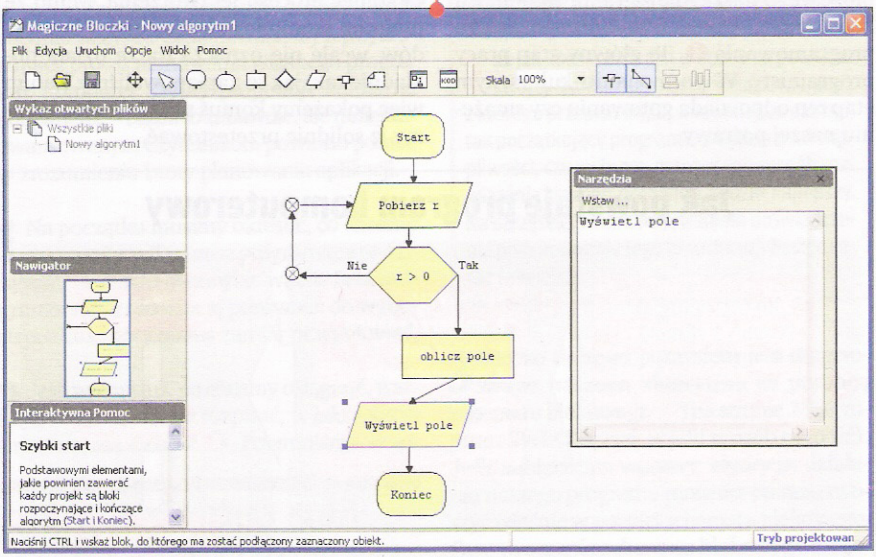
6 Uruchomienie aplikacji to jeszcze nie koniec procesu jej tworzenia. Mimo że program skompilował się wreszcie bez błędów, wcale nie oznacza to, że działa dokładnie tak, jak to zaplanowaliśmy. Zanim więc pokażemy komuś nasze dzieło, musimy je solidnie przetestować.

Tworzenie programu składa się z kilku etapów. Każdy jest bardzo ważny, więc żadnego nie należy pomijać

Jak tworzyć schematy blokowe

Schemat blokowy to język służący do opisywania algorytmu działania programu komputerowego w taki sposób, aby na jego podstawie można było napisać kod źródłowy tego programu w niemal dowolnym języku. Umiejętność tworzenia algorytmów i schematów blokowych nie jest więc związana jedynie z językiem C++, ale z pewnością przyda się nam w nauce innych języków programowania. Schematy blokowe najlepiej rysować na kartce papieru. Jednak do ich stworzenia możemy również wykorzystać aplikację Word (w tak zwanych autokształtach znajdują się odpowiednie figury) czy specjalnie do tego celu stworzoną darmową i prostą w obsłudze aplikację Magiczne Bloczki (znajdziemy ją na płycie dołączonej do tej książki).

1 Na początku schematu blokowego każdego programu umieszczamy figurę 1 (prostokąt z zaokrąglonymi rogami) oznaczającą początek algorytmu (czyli początek naszego programu).



2 Kolejne bloki naszego schematu zależą już tylko i wyłącznie od wykonywanego przez projektowany program zadania. Jeśli na przykład jego celem będzie obliczenie pola koła na podstawie wprowadzonego przez użytkownika promienia, następną figurą będzie równoległobok. Stosowany jest on do odczytu lub zapisu danych – w tym wypadku do pobrania z klawiatury wartości promienia 2.

► Komputer ŚWIAT radzi

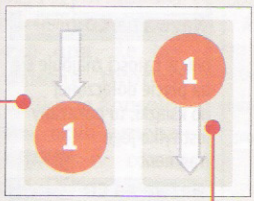
Kolejne figury łączymy ze sobą strzałką. W większości wypadków każda figura (blok) ma jedną strzałkę wchodzącą i jedną wychodzącą.

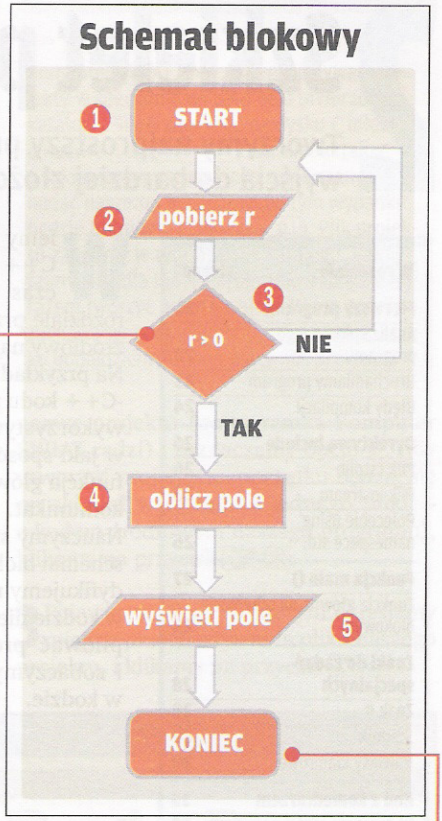
3 Po pobraniu wartości promienia warto sprawdzić, czy wprowadzona przez użytkownika liczba jest większa od zera. Jeśli nie jest, prosimy użytkownika o ponowne wprowadzenie promienia. Gdy wartość jest dodatnia, przechodzimy do dalszej części naszego programu. Blok, który reprezentuje opisaną sytuację (podjęcie decyzji) symbolizowany jest przez romb 3. Blok ten ma jedną strzałkę wchodzącą i dwie wychodzące (przy jednej wpisujemy słowo TAK – warunek ● został spełniony, przy drugiej NIE – warunek jest fałszywy).

4 Następnym krokiem jest obliczenie pola koła. Wszystkie obliczenia obrazujemy za pomocą prostokąta 4. Oczywiście wewnątrz niego nie wpisujemy dokładnych obliczeń, a jedynie krótkie hasło.

► Komputer ŚWIAT radzi

Nasze kolejne programy będą prawdopodobnie coraz bardziej rozbudowane. Jeśli więc schemat blokowy nie mieści się nam na jednej kartce papieru, możemy skorzystać z koła, które reprezentuje tak zwany łącznik stronicowy. Wystarczy doprowadzić do niego strzałkę z ostatniego bloku na jednej kartce i wpisać do niego dowolny symbol (liczbę czy literę) ●. Następnie na drugiej kartce trzeba umieścić takie samo kółko (z tym samym symbolem) ● i rozpocząć od niego dalszą część algorytmu. Uwaga! Łącznik stronicowy możemy wykorzystać również w ramach jednej strony – na przykład po to, aby nie krzyżować ze sobą strzałek.





5 Kolejną czynnością wykonywaną przez nasz program jest wyświetlenie obliczonego pola na ekranie. Rysujemy więc już znany nam blok wejścia-wyjścia 5 (równoległobok). Wewnątrz niego wpisujemy informację o wyświetleniu danych na ekranie naszego komputera.

6 Każdy stworzony przez nas algorytm powinien zakończyć się blokiem ●. Oznacza on pomyślne zakończenie działania naszego programu.

2 Szkielet programu

Tworzymy najprostszy program, który będzie punktem wyjścia do bardziej złożonych projektów

W rozdziale	strona
Pierwszy program	21
Analiza pierwszego programu	22
Uruchamiamy program	23
Błędy kompilacji	24
Dyrektywa include	26
Plik cstdlib	26
Plik iostream	26
Polecenie using namespace std;	26
Funkcja main ()	27
Funkcja główna na schemacie blokowym	28
Znaki do zadań specjalnych	28
Znak #	28
Średnik	28
Nawiasy klamrowe	29
Kod z komentarzem	30
Wyłączanie części programu	30
Co i jak komentować	31
Komentarz na schemacie blokowym	31

Wiemy już, że będziemy programować w języku C++ za pomocą aplikacji Dev-C++. Najwyższy czas napisać nasz pierwszy program. W tym rozdziale przeczytamy, w jaki sposób wygenerować kod źródłowy najprostszego programu.

Na przykładzie automatycznie wstawionego przez Dev-C++ kodu poznamy kolejne polecenia, które będziemy wykorzystywali w każdym projekcie. Dowiemy się, w jaki sposób dołączyć niezbędne biblioteki, co to jest funkcja główna oraz jak wyświetlić na ekranie dowolny komunikat.

Nauczymy się również komentować kod źródłowy oraz schemat blokowy. Dzięki temu w przyszłości łatwiej zmodyfikujemy nasz program. Przekonamy się też, że błędy w kodzie nie są wcale takie straszne – spróbujemy skompilować program z umyślnie wstawionym błędem i zobaczymy, jak radzić sobie z nieprawidłowościami w kodzie.

CD-ROM

plik znajduje się na płycie dołączonej do książki

Nazwa na CD

plik w całości znajduje się na płycie dołączonej do książki, tu pokazany jest tylko jego fragment

Nazwa na CD

Pierwszy program

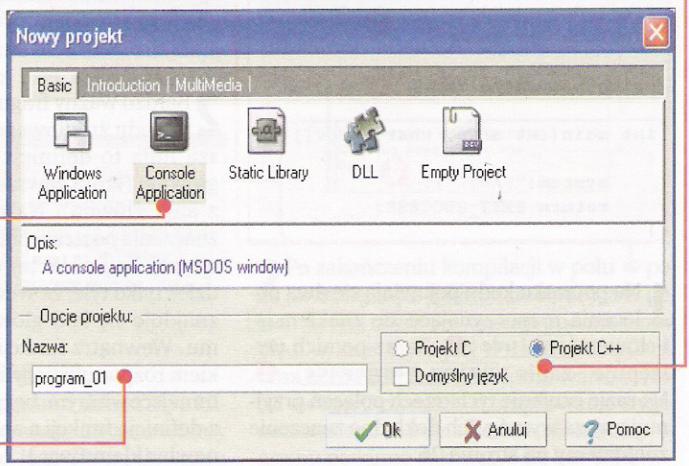
Nie znamy jeszcze żadnej konstrukcji języka C++. Nie ma jednak sensu wkuwać teorii. Nasz pierwszy program w C++ stworzymy za pomocą kreatora

Potrafimy już zaplanować pracę nad naszym programem oraz naszkicować schemat blokowy. Przyszedł czas na napisanie kodu źródłowego pierwszego programu. Ustalmy więc, że jego zadaniem będzie wyświetlanie dowolnego komunikatu. Co prawda efekt działania naszej aplikacji nie będzie zbyt spektakularny, ale dzięki takiemu prostemu przykładowi zobaczymy, jak wygląda szkielet kodu źródłowego w języku C++.

1 Na początek uruchamiamy aplikację Dev-C++.

2 Następnie tworzymy w niej nowy projekt (z menu Plik wybieramy podmenu Nowy i pozycję Projekt...). Pojawia się okno, w którym klikamy na ikonę (oznacza ona, że będziemy tworzyć aplikację konsolową, czyli uruchamianą w oknie MS-DOS-u). W pole wpisujemy nazwę projektu (patrz ramka Komputer ŚWIAT radzi) i zaznaczamy opcję . Aby utworzyć projekt (na początku będzie się on składał wyłącznie z jednego pliku – z kodem źródłowym naszego programu), klikamy na przycisk OK.

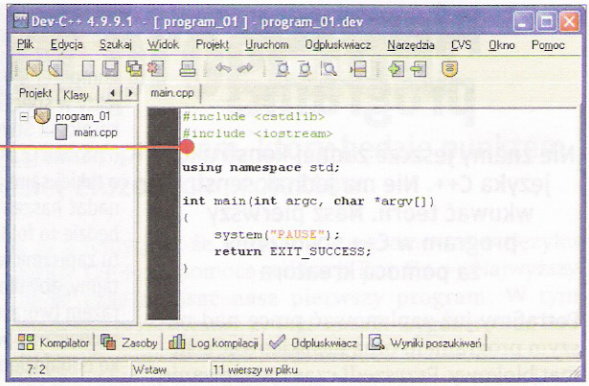
3 Pojawia się okno, w którym określamy miejsce przechowywania pliku naszego projektu, i klikamy na przycisk Zapisz.



► Komputer ŚWIAT radzi

Komputer ŚWIAT poleca przechowywać projekty w specjalnie do tego celu utworzonym katalogu. Stwórzmy więc na dysku C folder o nazwie biblioteczka, a w nim katalog o takiej samej nazwie, jaką zamierzamy nadać naszemu projektowi (w tym wypadku będzie to folder program_01). Plik projektu zapiszmy w tym właśnie katalogu. Pamiętajmy, aby dla innych programów za każdym razem tworzyć nowe katalogi. W ten sposób zachowamy porządek na dysku i później szybko odnajdziemy interesujący nas program.

4 Nowy projekt jest już stworzony, a w oknie edycji kodu źródłowego pojawia się szkielet naszej aplikacji (jest automatycznie generowany przez program Dev-C++). Zapiszmy od razu plik z kodem źródłowym – z menu Plik wybieramy pozycję Zapisz wszystko i w oknie, które się otwiera, klikamy na przycisk Zapisz. Plik z kodem źródłowym o nazwie main.cpp pojawi się w tym samym katalogu, w którym zapisaliśmy plik projektu.



Analiza pierwszego programu

Jeśli pierwszy raz spotykamy się z kodem źródłowym programu w C++, z pewnością trudno nam zrozumieć występujące w nim polecenia. Warto więc przyjrzeć się stworzonemu automatycznie szkieletowi programu. Znaczenie kolejnych zapisów poznamy w dalszej części tego rozdziału.

```
#include <cstdlib>          1
#include <iostream>

using namespace std;        2

int main(int argc, char *argv[])   3
{
    system("PAUSE");        4
    return EXIT_SUCCESS;
}
```

1 Na początku kodu pojawiają się dwa polecenia rozpoczynające się znakiem # i słowem include 1. Zaraz po nich występuje zapis using namespace 2. Na razie istnienie tych trzech poleceń przyjmujemy za wymóg. Ich dokładne znaczenie znajdziemy na stronie 26.

Komputer ŚWIAT radzi

Dla ułatwienia pracy programisty edytor Dev-C++ koloruje kod źródłowy. Na przykład wszystkie polecenia zaczynające się od znaku # oznaczone są kolorem zielonym, znaki umieszczone między cudzysłowami kolorowane są na czerwono. Warto również zauważyć, że wszystkie słowa kluczowe (wyrazy, które mają szczególne znaczenie w C++) zostają pogrubione. Dzięki kolorowaniu kodu jest on bardziej przejrzysty.

2 Bardzo ważny fragment naszego szkieletu kodu źródłowego to linijki 3. Pierwsza linia to definicja tak zwanej funkcji głównej (która zawsze nazywa się main – z ang. główna). Na razie nie dociekamy znaczenia poszczególnych elementów omawianej linijki. W tej chwili musimy wiedzieć tylko tyle, że wewnątrz funkcji main znajduje się kod główny naszego programu. Wewnątrz funkcji – czyli między znakiem rozpoczynającym nawias klamrowy { umiejscowionym bezpośrednio po linijce z definicją funkcji a znakiem zamykającym nawias klamrowy }.

► Komputer ŚWIAT radzi

Warto pamiętać, że wnętrze funkcji (czyli to, co znajduje się między definicją funkcji wraz ze znakiem { a znakiem } bardzo często jest nazywane ciałem funkcji.

3 Jak już wiemy, główny kod naszego programu znajduje się wewnątrz funkcji `main`. Jakie więc polecenia zostały w tym miejscu wstawione automatycznie przez aplikację Dev-C++? Pierwsze ❹ nie jest charakterystyczne dla każdego programu źródłowego, a jedynie dla aplikacji konsolowych uruchamianych w Windows. Ponieważ nasz program będzie uruchamiany w oknie MS-DOS-u (jak pamiętamy, stworzyliśmy projekt aplikacji konsolowej), musimy sprawić, aby po zakończeniu wykonywania wszystkich poleceń okno nie zostało od razu zamknięte. Polecenie `system("PAUSE");` nakazuje więc systemowi wstrzymać zamknięcie okna z efektami pracy naszego programu.

4 Ostatnie polecenie funkcji `main` (`return EXIT_SUCCESS;`) informuje system, że wykonanie instrukcji znajdujących się wewnątrz niej zakończyło się sukcesem.

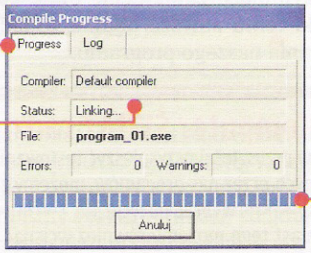
► Komputer ŚWIAT radzi

Dwa ostatnie polecenia, ze względu na zadanie, które wykonują, powinny się zawsze znajdować na końcu funkcji `main`, czyli tuż przed znakiem } oznaczającym koniec ciała funkcji głównej.

Uruchamiamy program

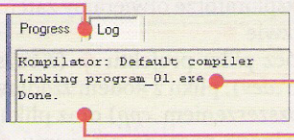
Szkielet naszego programu składa się z kilku linijek kodu. Zapewne zastanawiamy się, co się stanie po jego skompilowaniu i uruchomieniu. Oczywiście na wiele liczyć nie można, ale na pewno warto to sprawdzić.

1 Na początku skompilujemy kod źródłowy. Aby to zrobić, z menu Uruchom wybieramy pozycję Kompiluj. Na ekranie pojawia się okno ❶, w którym pasek ❷ informuje o postępie kompilacji.



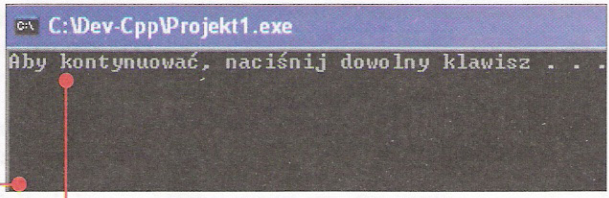
► Komputer ŚWIAT radzi

Jeśli po zakończonej kompilacji klikniemy na zakładkę ❸, będziemy mogli zobaczyć log kompilacji ❹. Łatwo możemy zauważyć, że wraz z kompilacją został wykonany proces łączenia (linkowania) i w efekcie powstał plik wykonywalny naszego programu ❺.



2 Po zakończeniu kompilacji w polu ❸ pojawia się napis Done. (zrobiono), a w polach Errors: i Warnings: zostaje podana liczba błędów, które wystąpiły w trakcie kompilacji, oraz liczba ostrzeżeń, na które warto zwrócić uwagę. Oczywiście skompilowanie wygenerowanego automatycznie szkieletu programu powinno zakończyć się bez błędów.

3 Po skompilowaniu programu możemy go uruchomić. Aby to zrobić, z menu Uruchom wybieramy pozycję Uruchom. Na ekranie pojawia się okno MS-DOS-u, a w nim komunikat. Wywołało go polecenie `system("PAUSE");` wstrzymujące zamknięcie okna konsoli. Dzięki temu zawsze będzie widać efekt działania naszego programu.



► Komputer ŚWIAT radzi

Aby w przyszłości przyspieszyć testowanie napisanych przez nas progamów, możemy z menu Uruchom wybrać pozycję Kompiluj i uruchom (zamiast tego możemy również wcisnąć klawisz F9). Program zostaje skompilowany i od razu uruchomiony (jeśli w kodzie nie było błędów).

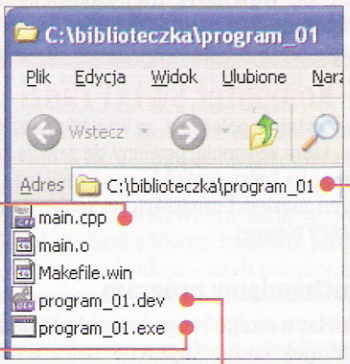
Błędy kompilacji

Nasz pierwszy program skompilował się bez błędów. Jednak musimy się przyzwyczaić do myśli, że prawie nigdy nie będzie tak dobrze i przed zobaczeniem efektów naszego programu będziemy musieli poświęcić sporo czasu na usunięcie błędów z kodu źródłowego. Warto więc już teraz zobaczyć, co robić, jeśli kompilator zakomunikuje nam, że kompilacja z powodu występujących błędów nie została zakończona sukcesem. Błędom Komputer ŚWIAT poświęcił również dodatek, który znajdziemy na stronie 122.

▼ Pliki wygenerowane przez kompilator

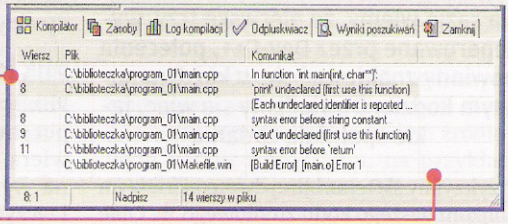
Po skompilowaniu naszego programu warto otworzyć katalog, w którym zapisaliśmy plik z kodem źródłowym. W Eksploratorze otwieramy zatem folder o nazwie.

Oprócz pliku projektu (z rozszerzeniem *.dev*), pliku z kodem źródłowym (z rozszerzeniem *.cpp*) oraz pliku wykonywalnego naszego programu (rozszerzenie *.exe*) w folderze znajdują się również dwa inne pliki. Plik main.o to tak zwany plik binarny skompilowany – jest efektem pracy kompilatora (wiele kompilatorów nadaje mu rozszerzenie *.obj*). Plik Makefile.win to zbiór reguł dla kompilatora, wykorzystywanych przy kompilacji programu. Dzięki niemu kompilator poradzi sobie z programem, którego kod źródłowy umieścimy w kilku różnych plikach.



► Komputer ŚWIAT radzi

Zapewne w przyszłości w naszym kodzie źródłowym nieraz będzie się pojawiało więcej błędów ●. Aby podejrzeć kolejne linijki kodu, w których kompilator wykrył nieprawidłowości, wystarczy dwukrotnie kliknąć na komunikat w oknie ●. Pamiętajmy jednak, że wiele komunikatów będzie następstwem błędów występujących we wcześniejszych liniach kodu. Jeśli więc poprawimy pierwszy błąd i ponownie skompilujemy program – być może wiele ostrzeżeń i błędów występujących w dalszej części kodu zniknie.



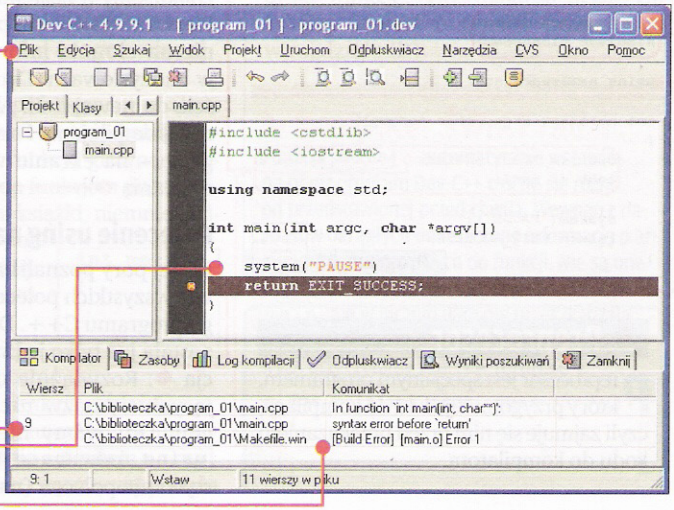
1 Na początku wprowadzimy drobną zmianę do naszego kodu – usuwamy na przykład średnik znajdujący się na końcu linii `system("PAUSE");`.

2 Włączamy kompilację kodu (z menu Uruchom wybieramy Kompiluj).

3 Tym razem nie pojawi się okno informujące o zakończonej sukcesem kompilacji. Zamiast tego zobaczymy taki widok ●.

4 W oknie ● wyświetlone zostały wszystkie komunikaty kompilatora. Przyjrzyjmy się szczególnie temu, w którym podany został numer wiersza z odnalezionym błędem ●. Wiersz ten został również oznaczony w oknie z kodem źródłowym ●. Jak już pewnie zauważyliśmy, kompilator znalazł błąd w linii następnej w stosunku do tej, w której usunęliśmy średnik. Pamiętajmy więc, że często błędów powinniśmy szukać w liniach znajdujących się przed tą, której nie rozumie kompilator.

5 Wystarczy już tylko usunąć błąd (wpisać usunięty wcześniej średnik) i ponownie skompilować program. Tym razem kompilacja powinna zakończyć się pomyślnie.



Dyrektywa include

Na razie wiemy, że pierwsze trzy, wygenerowane przez Dev-C++, polecenia powinny znajdować się w każdym naszym kodzie. Dowiedzmy się więc, jakie spełniają zadanie

Przyjrzyjmy się teraz bliżej dwóm pierwszym linijkom naszego kodu źródłowego. Obie zaczynają się od znaku `#`, po którym występuje słowo `include` (z ang. dołącz). W języku C++ zapis `#include` to tak zwana dyrektywa preprocesora – informacja dla preprocesora, aby w tym miejscu dołączył do naszego programu zawartość wskazanego pliku (czyli najpierw pliku o nazwie `cstdlib`, a później pliku o nazwie `iostream`).

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << "Nasz pierwszy program w C++";
    cout << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_01

Preprocesor

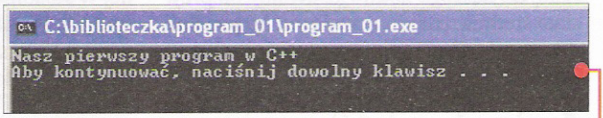
Preprocesor jest specjalnym programem, który przygotowuje kod do kompilacji, czyli zajmuje się nim przed przekazaniem kodu do kompilatora.

Plik cstdlib

W pliku `cstdlib` znajdują się definicje różnych użytecznych funkcji, które możemy wykorzystać w naszym programie. Na razie nie będziemy ich używać, ale nie usuwamy dyrektywy wstawiającej ten plik.

Plik iostream

Plik `iostream` (jest to skrót od input-output stream – strumień wejścia-wyjścia) zawiera definicje poleceń pozwalających na wyświetlanie informacji na ekranie i wczytywanie do programu danych z klawiatury. Dzięki `iostream` możemy na przykład wyświetlić na ekranie dowolny komunikat. Jak to zrobić? Wewnątrz funkcji głównej umieszczamy na przykład dwa polecenia. Ich dokładne wyjaśnienie znajdziemy w następnym rozdziale książki. Pierwsze polecenie powoduje wyświetlenie na ekranie informacji umieszczonej w cudzysłowach. Drugie powoduje przejście do nowej linii. Aby się o tym przekonać, kompilujemy i uruchamiamy nasz program – na ekranie widzimy wynik jego działania.



Polecenie using namespace std;

Do tej pory poznaliśmy już znaczenie prawie wszystkich poleceń ze szkieletu naszego programu C++. Do rozszyfrowania pozostała już tylko jedna tajemnicza instrukcja. Rozwiązuje ona problem dublujących się nazw różnych funkcji i poleceń. Gdybyśmy nie wpisali instrukcji `using namespace std;`, przy wykorzystywaniu poleceń (na przykład wspomnia-

Nazwa dołączanego pliku

W wielu złożonych programach możemy spotkać się z nieco odmiennym zapisem dyrektywy `#include`, w której nazwa dołączanego pliku umieszczana jest w cudzysłowach, a nie w nawiasach ostrych. Różnica w działaniu obu dyrektyw jest taka, że w wypadku nawiasów ostrych dołączany plik poszukiwany jest w katalogu, w którym kompilator przechowuje różne tego typu pliki (w wypadku aplikacji Dev-C++ jest to katalog C:\Dev-Cpp\include). Jeśli nazwę zapiszemy w cudzysłowach, plik będzie poszukiwany w folderze bieżącym (tam gdzie zapisany jest plik z kodem źródłowym naszego programu, na przykład C:\biblioteczka\program_01).

```
#include <iostream>
#include "wyswietlanie.h"
```

nego już `cout` czy `endl`) musielibyśmy wskazywać kompilatorowi, skąd one pochodzą (czyli zamiast `cout` musielibyśmy napisać `std::cout`, a zamiast `endl` wpisać `std::endl`). `std` oznacza bibliotekę standardową, w której znajdują się definicje wszystkich najważniejszych symboli oraz poleceń i funkcji. Będziemy z nich często korzystać, więc warto za pomocą polecenia `using namespace` ułatwić sobie pracę.

Funkcja main ()

Najważniejszym elementem kodu programu jest funkcja główna. Dowiedzmy się o niej czegoś więcej

Wiemy już (z części poświęconej analizie szkieletu programu w C++), że każdy program musi zawierać funkcję główną o nazwie `main`. Co prawda funkcjom poświęcony został cały rozdział książki, niemniej jednak warto już teraz poznać szczegóły budowy tego ważnego elementu kodu źródłowego.

```
int main()
{

}
```

Najprostsza definicja funkcji głównej ma postać:

- `main` jest nazwą funkcji głównej,
- nawiasy okrągłe oznaczają, że mamy do czynienia właśnie z funkcją,
- nawiasy klamrowe wyznaczają ciało funkcji; pomiędzy nimi umieszczamy instrukcje, które funkcja ma wykonać,
- `int` oznacza typ wyniku zwracanego przez funkcję; `int` oznacza liczbę całkowitą (więcej o typach przeczytamy w rozdziale III).

Komputer ŚWIAT radzi

Funkcja `main` automatycznie wstawiona przez program Dev-C++ trochę się różni od przedstawionej przed chwilą. Wewnątrz nawiasów okrągłych umieszczono informację o argumentach wejściowych do funkcji. Nie są one co prawda obowiązkowe w wypadku funkcji `main`, ale w niczym nam nie przeszkadzają, więc nie kasujmy ich. O argumentach wejściowych przeczytamy więcej w rozdziale V.

```
int main(int argc, char *argv[])
{

}
```

Funkcja główna na schemacie blokowym

Wykonywanie naszego programu rozpoczyna się od funkcji `main`, a kończy wraz z ostatnim poleceniem występującym w ciele funkcji głównej. Dlatego na schemacie blokowym funkcja `main` reprezentowana jest przez dwa prostokąty z zaokrąglonymi bokami 1, które rozpoczynają i kończą działanie algorytmu programu.

Polecenie return

Funkcja `main` po zakończeniu swojego działania zwraca liczbę całkowitą (określa to słowo `int` umieszczone przed nazwą funkcji). Dlatego ważne jest umieszczenie w ciele funkcji głównej, na samym jej końcu, polecenia `return`, po którym znajduje się zapis 2. Jest on równoważny liczbie 0. Jeśli funkcja `main` zostanie wykonana do końca, zwróci liczbę 0, co oznacza, że zakończyła się sukcesem. Jeśli działanie funkcji zostanie przerwane wcześniej, zwrócona wartość będzie większa od 0 – taka sytuacja oznacza dla systemu informację, że w funkcji głównej znajdują się błędy.

```
return EXIT_SUCCESS;
```

Znaki do zadań specjalnych

Kod programu to ciąg instrukcji i różnych znaków. Niektóre z nich pełnią szczególnie ważną rolę. Komputer ŚWIAT przedstawia te, na które trzeba zwrócić uwagę

Jak już z pewnością zauważyliśmy, niektóre znaki mają w kodzie specjalne znaczenie. Komputer ŚWIAT w jednym miejscu zebrał te najważniejsze. Warto się z nimi zapoznać i dobrze przyswoić ich zastosowanie. Pamiętamy przecież, że programowanie wymaga niesamowitej precyzji i brak lub złe zastosowanie nawet jednego znaku, może podczas kompilacji wywołać lawinę błędów.

Znak

Dowiedzieliśmy się już wcześniej, że znak # występuje przed dyrektywami preprocesora, czyli informacjami dla programu, który przygotowuje kod źródłowy dla kompilatora. W Dev-C++ dyrektywy preprocesora są oznaczane w kodzie kolorem zielonym 1.

Średnik

Średnik w C++ jest znakiem szczególnym. Występuje on najczęściej na końcu linii, a jego obecność powoduje, że polecenie znajdujące się przed nim staje się instrukcją. Jeśli nie jesteśmy jeszcze pewni, kiedy powinniśmy stawiać średnik, nie martwmy się. Po przestudiowaniu kilku przykładów kodów źródłowych będziemy bezbłędnie posługiwać się średnikiem. Niemniej jednak warto na ten znak zwracać szczególną uwagę – jego brak jest częstą przyczyną błędów w kodzie.

Nawiasy klamrowe

Nawiasy klamrowe umożliwiają tworzenie bloku instrukcji, który przez kompilator traktowany jest tak, jak jedna instrukcja. Za pomocą nawiasów klamrowych możemy więc łączyć wiele instrukcji prostych tak, aby zawsze były wykonywane razem. Przed blokiem instrukcji zazwyczaj znajduje się słowo kluczowe, które zdradza przeznaczenie takiego bloku instrukcji – może to być na przykład nazwa funkcji głównej ❷ lub na przykład słowo ❸ (używane do tworzenia pętli – więcej na temat pętli w rozdziale IV).

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int i;
    for (i=0;i<10;i++)
    {
        cout << i;
        cout << endl;
    }

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

❶ ❷ ❸

Możemy tworzyć także bloki instrukcji bez stosowania żadnego słowa kluczowego i w ten sposób wydzielać fragmenty kodu źródłowego. Później dowiemy się, w jakich przypadkach stosuje się takie konstrukcje.

► Komputer ŚWIAT radzi

Pamiętajmy, aby w kodzie nawiasy (okrągłe, klamrowe czy kwadratowe) występowały zawsze parami. Komputer ŚWIAT radzi, aby po wstawieniu w kodzie nawiasu otwierającego od razu wstawiać nawias zamykający, i dopiero wtedy między tymi dwoma nawiasami umieszczać na przykład instrukcje. Dzięki temu nie zapomnimy nigdy o umieszczeniu nawiasu zamykającego.

► Komputer ŚWIAT radzi

Nie ma przeszkód, aby w bloku instrukcji wydzielonym za pomocą nawiasów klamrowych znajdował się inny blok instrukcji (lub wiele bloków). W bardziej rozbudowanych programach, takich zagnieżdżonych w sobie bloków będzie bardzo dużo. Dlatego warto zapoznać się z informacjami z ramki Ładny kod.

▼ Ładny kod

Zwróćmy uwagę na sposób zapisywania kodu – instrukcje w każdym wydzielonym bloku przesunięte są w prawo ● (za pomocą klawisza tab). Warto stosować taką zasadę – wtedy kod wygląda niezwykle przejrzyście i bardzo łatwo jest go analizować. Wielu programistów stosuje również trochę inny format zapisu bloków instrukcji. Polega on na umieszczeniu nawiasu klamrowego rozpoczynającego taki blok w tej samej linii co słowo kluczowe zdradzające przeznaczenie tego bloku ●.

```
int main(int argc, char *argv[]){
    system("PAUSE");
    return 0;
}
```

Kod z komentarzem

Kod może być tak długi i skomplikowany, że trudno nam będzie przypomnieć sobie, jakie zadanie miał spełniać każdy z jego fragmentów. Nauczmy się więc pisać komentarze

Wyobraźmy sobie, że stworzyliśmy kilkusetlinijkowy kod i po miesiącu chcemy go udoskonalić. Niemal pewne jest, że będziemy musieli od nowa go przeanalizować, aby przypomnieć sobie, jakie zadania spełniały jego poszczególne fragmenty (sytuacja może być jeszcze trudniejsza, gdy przyjdzie nam poprawiać kod innego programisty). Dlatego Komputer ŚWIAT poleca komentowanie kodu źródłowego. Jeśli komentarz umieścimy zgodnie z poniższymi wskazówkami, nie będzie on widoczny dla kompilatora i w żaden sposób nie wpłynie na działanie i wielkość naszego programu.

1 Pierwszym sposobem skomentowania fragmentu kodu źródłowego jest umieszczenie komentarza pomiędzy znakiem otwarcia komentarza a znakiem jego zakończenia. Taki komentarz może obejmować kilka linijek kodu. Dev-C++ komentarz koloruje na niebiesko.

2 Drugi sposób umożliwia umieszczenie komentarza w jednej linii. W jej dowolnym miejscu wystarczy umieścić dwa znaki slash, a wszystko, co znajduje się po nich, aż do końca linijki, staje się komentarzem. Taki komentarz jest w Dev-C++ również kolorowany na niebiesko.

► Komputer ŚWIAT radzi

Komentowanie kodu za pomocą podwójnego slasha nie jest rozpoznawane przez starsze kompilatory. Dlatego lepiej jest stosować sposób opisany w punkcie 1.

Wyłączanie części programu

Wykorzystując komentarze, możemy na chwilę wyłączyć część programu. Takie rozwiązanie powoduje, że kompilator nie widzi instrukcji umieszczonych pomiędzy znakami początku i końca komentarza. Chwilowe wyłączenie fragmentu kodu mo-

```
/* Program wyświetla na ekranie
fragment tekstu.
Autor: Jacek Marcinkowski
Data: 15.03.2005 */

#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << "Nasz pierwszy program w C++";
/*
    cout << endl;
*/

    system("PAUSE");   //blokujemy zamknięcie okna
    return EXIT_SUCCESS;
}
```

że nam się przydać podczas testowania czy poszukiwania błędów. W ten sposób można skoncentrować się i dokładnie sprawdzić działanie wybranej części programu. Pamiętajmy jednak, aby nie wyłączać w ten sposób zbyt długich fragmentów kodu.

Co i jak komentować

Jakość komentarzy w kodzie źródłowym świadczy o programiście. Oczywiście najgorszym rozwiązaniem jest całkowity brak dodatkowych opisów. Równie niekorzystnym zjawiskiem jest tworzenie komentarzy, które nie niosą żadnych informacji, na przykład (wiadomo, że instrukcja powoduje wyświetlenie na ekranie słowa znajdującego się w cudzysłowach, więc komentarz jest całkowicie zbędny). Ale jak znaleźć złoty środek? Niestety, właściwe komentowanie wymaga takiej samej wprawy jak programowanie.

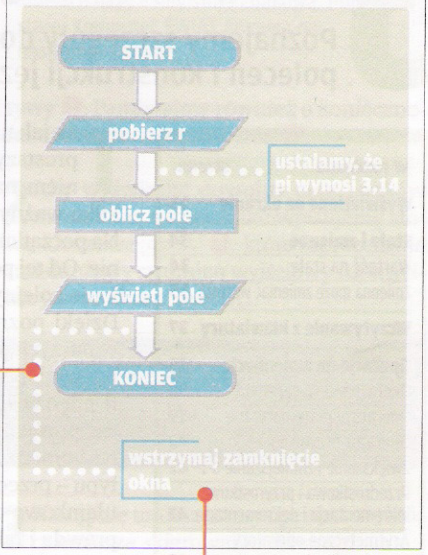
```
cout << "Witaj!"; //wyświetlamy słowo Witaj!
```

Komentarz na schemacie blokowym

Nie tylko w kodzie warto wpisywać notatki, które ułatwią nam później wszelkie modyfikacje. Komentarze powinniśmy umieszczać również na schemacie blokowym, który tworzymy przed zakodowaniem programu. Co to jest schemat blokowy oraz w jaki sposób go stworzyć, dowiedzieliśmy się pod koniec pierwszego rozdziału. Teraz nauczymy się umieszczać komentarze na schemacie.

1 Z miejsca, w którym chcemy umieścić komentarz, prowadzimy przerywaną linię do najbliższego wolnego obszaru schematu.

Komentarze na schemacie



2 Następnie rysujemy niedokończony prostokąt, wewnątrz którego umieszczamy komentarz. Pamiętajmy, aby figury komentarza nie przeszkadzały w zrozumieniu schematu blokowego.

Komputer ŚWIAT radzi

Nie ma sensu komentować każdego bloku na schemacie blokowym. Najlepiej umieszczać wyjaśnienia tylko bardziej złożonych zadań, które ma wykonać program. W postaci komentarzy możemy także na schemacie umieścić definicje stałych liczb, z których korzystamy w programie (na przykład liczby pi) lub dodatkowych niewielkich zadań, których nie warto rozbijać na oddzielne bloki, a które powinien realizować nasz progam.

3 Podstawy języka C++

Poznajemy szczegóły dotyczące najważniejszych poleceń i konstrukcji języka C++

W rozdziale	strona
Wyświetlone na ekranie	**33**
Stałe i zmienne	**34**
Wartość na stałe	34
Zmienna może zmieniać wartość	35
Wczytywanie z klawiatury	**37**
Operacje na zmiennych	**39**
Operatory a wyrażenia	39
Operatory matematyczne	39
Operator przypisania	40
Zwiększanie i zmniejszanie o 1	41
Przedrostkowa i przyrostkowa inkrementacja i dekrementacja	42
Arytmetyczne operatory przypisania	43
Przypisanie ze zmianą wartości o 1	44
Odejmowanie a znak minusa	45
Relacje między zmiennymi	45
Nasz pierwszy przydatny program	**46**
Określamy cel	46
Piszemy algorytm	46
Schemat blokowy	47
Piszemy aplikację	48
Typy danych	**48**
Podział typów	49
Liczby całkowite	49
Liczby z przecinkiem	50
Typ znakowy	52
Prawda czy fałsz	52
Inne typy	53
Pamięciożerne typy	53

Tak jak naukę języka obcego należy rozpocząć od najprostszych zwrotów, tak przygodę z programowaniem najlepiej rozpocząć od dokładnego poznania kilku ważnych instrukcji języka C++.

Na początku nauczymy się wyświetlać informacje na ekranie. Od tej pory będziemy mogli naocznie sprawdzać działanie kolejnych konstrukcji językowych.

Dzięki poznaniu zmiennych i stałych będziemy mogli w naszych programach różnym wartościom przypisywać nazwy. Operatory pozwolą nam tworzyć wyrażenia, czyli między innymi dodawać, mnożyć, zmieniać wartość czy porównywać ze sobą zmienne i stałe.

Na koniec nauczymy się tworzyć zmienne i stałe różnego typu – przechowujące liczby całkowite, liczby z wartością ułamkową oraz takie, które mogą mieć dwie wartości – prawda i fałsz.

▼ CD-ROM

plik znajduje się na płycie dołączonej do książki

Nazwa na CD

plik w całości znajduje się na płycie dołączonej do książki, tu pokazany jest tylko jego fragment

Nazwa na CD

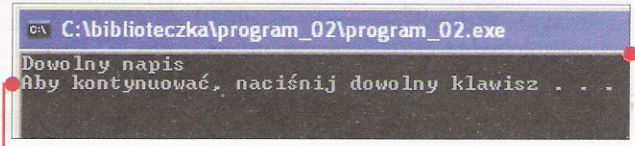
Wyświetlone na ekranie

Naukę programowania najlepiej rozpocząć od poznania sposobu wyświetlania informacji na ekranie. Dzięki tej wiedzy będziemy mogli sprawdzać działanie kolejnych poleceń

Szczegółowe omawianie elementów programowania w języku C++ najlepiej rozpocząć od instrukcji pozwalającej na wyświetlanie informacji na ekranie. Wiedza ta bardzo szybko nam się przyda – będziemy mogli na własne oczy zobaczyć działanie większości omawianych przez Komputer ŚWIAT instrukcji.

► Komputer ŚWIAT radzi

Dowolny ciąg znaków umieszczony w cudzysłowach nazywamy stałą znakową. Pojęcie stałej Komputer ŚWIAT omawia na następnej stronie.




```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << "Dowolny napis";
    cout << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_02

1 Najpopularniejszym sposobem wyświetlania informacji tekstowych w C++ jest tak zwany strumień wyjściowy `cout` (czytamy si-ałt) – występował już w kilku przykładach z poprzedniego rozdziału. Definicja polecenia `cout` znajduje się w pliku `iostream`, który jak pamiętamy, dołączamy do naszego programu za pomocą dyrektywy ❶. Pamiętajmy również o konieczności umieszczenia polecenia ❷.

2 Aby wyświetlić dowolny ciąg znaków na ekranie, na początku wpisujemy słowo kluczowe `cout` ❸, które reprezentuje standardowy strumień wyjściowy (domyślnie jest to ekran konsoli).

3 Następnie umieszczamy operator ●, który oznacza skierowanie do strumienia.

4 Teraz wystarczy już tylko określić, co ma zostać skierowane do strumienia wyjściowego. Aby na przykład wyświetlić dowolny napis, wystarczy wpisać go w cudzysłowach (na przykład ●).

5 Na koniec instrukcji wpisujemy średnik. Po skompilowaniu oraz uruchomieniu naszego kodu na ekranie monitora widzimy ●.

▼ Znak końca linii

Jeśli do strumienia `cout` skierujemy symbol `endl`, spowoduje to przejście na ekranie do nowej linii ● (`endl` jest synonimem wciśnięcia klawisza enter).

Stałe i zmienne

Stałe i zmienne będą występować prawie we wszystkich naszych programach. Warto je więc poznać

Wyobraźmy sobie taką sytuację: mamy do napisania prośbę do naszego szefa. Jak wiadomo, prośba powinna mieć miły ton, więc co kilka wyrazów umieszczamy w niej zwrot „Szanowny Panie Prezesie". Gdybyśmy mieli pieczątkę z takim napisem, stworzenie pisma do szefa byłoby znacznie łatwiejsze. W programowaniu odpowiednikiem pieczątki są stałe i zmienne. Stałe i zmienne mają nawet pewną zaletę w stosunku do prawdziwych pieczątek – dużo łatwiej jest je stworzyć. Zobaczmy, do czego może nam się przydać stała i zmienna.

Wartość na stałe

Stwórzmy w Dev-C++ nowy projekt (aplikację konsolową) – możemy zapisać go w katalogu program_03 pod taką samą nazwą.

1 Wewnątrz funkcji `main` umieśćmy definicję stałej. Aby to zrobić, wpisujemy słowo kluczowe `const`. Oznacza ono, że za chwilę będziemy definiować stałą.

2 Teraz powinniśmy określić typ stałej, czyli poinformować kompilator, jakiego rodzaju dane chcemy w stałej przechowywać. Więcej o typach danych dowiemy się na stronach 48–53. W tej chwili wystarczy nam wiedzieć, że aby do stałej zapisać dowolny ciąg znaków, użyjemy typu `string`.

►Komputer ŚWIAT radzi

`string` co prawda nie jest wbudowanym w C++ typem danych, ale należy do biblioteki standardowej języka C++, która jest dołączana do każdego naszego programu za pomocą dyrektywy `#include`. Więcej o typie `string` dowiemy się ze strony 103.

3 Kompilator już wie, że będziemy chcieli zadeklarować stałą typu `string`. Podajmy więc jej nazwę (identyfikator) (na przykład ●) oraz po znaku równości jej wartość (na przykład ●) – pamiętajmy, że ciąg znaków wpisujemy w cudzysłowach. Nie można oczywiście zapomnieć o średniku na końcu wpisanej przez nas instrukcji.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    const string MK = "Marek Kowalski";
    cout << "Nazywam sie " << MK << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_03

4 Stała jest już zdefiniowana – możemy więc używać jej tak jak pieczątki. Jeśli na przykład wpiszemy taką instrukcję ●, po skompilowaniu i uruchomieniu kodu na ekranie pojawi się widok ●. Jak możemy łatwo zauważyć, wstawiony przez nas identyfikator stałej został zamieniony na odpowiadającą mu wartość ●. Oczywiście identyfikatora możemy w kodzie naszego programu używać wielokrotnie.

```
C:\biblioteczka\program_03.exe
Nazywam sie Marek Kowalski
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Wiele danych na ekran

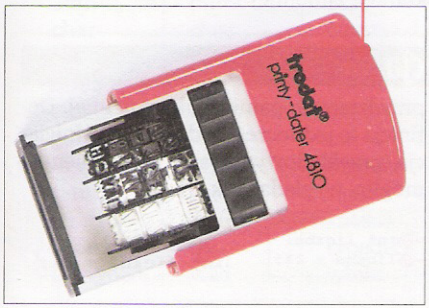
Słowo kluczowe `cout` reprezentuje strumień wyjściowy, a więc można do niego skierować więcej niż jeden napis. Spójrzmy na przykład na instrukcję. Należy ją rozumieć w taki sposób: Do standardowego strumienia wyjściowego (ekranu) wyślij napis Hej!, następnie wyślij symbol `endl` (który oznacza przejście do nowej linii), później skieruj na ekran napis Tu Twoj komputer i na końcu ponownie symbol przejścia do nowej linii. Efekt działania takiego kodu będzie taki.

```
Hej!
Tu Twoj komputer
```

```
cout << "Hej!" << endl << "Tu Twoj komputer" << endl;
```

Zmienna może zmieniać wartość

Wiemy już, co to jest i w jaki sposób korzystać ze stałej. Ten element języka ma jednak zasadniczą wadę – stała, jak sama nazwa wskazuje, jest zawsze taka sama. Stałej można przypisać wartość tylko raz. Można ją więc porównać do pieczątki – raz wyrobiona, przez całe swoje życie stawia ten sam odcisk. Ale przecież są też i takie pieczątki, których treść możemy zmodyfikować – przykładem może być pieczęć z ustawialną datą.



Odpowiednikiem takiej pieczęci w programowaniu jest zmienna. Porównanie jest tym lepsze, że wartość zmiennej, tak samo jak wspomnianej pieczęci, możemy modyfikować tylko w określonym zakresie, zależnym od typu zmiennej (o tym więcej na stronach 48–53).

Zobaczmy więc, w jaki sposób korzystać ze zmiennych.

1 Tworzymy nowy projekt (nazwijmy go na przykład program_04).

2 Na początku musimy zadeklarować zmienną, czyli określić jej typ oraz identyfikator. W tym celu wpisujemy nazwę typu (na przykład `int`, który określa, że zmienna może przechowywać liczbę całkowitą) oraz nazwę (identyfikator zmiennej) – na przykład `liczba1` ❶. Więcej o typach danych przeczytamy na stronach 48–53. Oczywiście na końcu stawiamy średnik.

3 Po zadeklarowaniu zmiennej możemy ją zainicjować, czyli pierwszy raz przypisać jej wartość. Jak to zrobić? Bardzo łatwo – wystarczy wpisać identyfikator zmiennej, znak równości i wartość, którą jej chcemy przypisać ❷. No i oczywiście średnik.

Miejsce w pamięci

Stałe i zmienne to tak naprawdę określone miejsca w pamięci komputera. Dlatego stworzenie stałej lub zmiennej to przypisanie do jej identyfikatora określonego adresu pamięci, pod którym znajduje się wartość tej stałej lub zmiennej.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int liczba1;1
    liczba1 = 21;2
    cout << "Zmienna liczba1 ma wartosc: " << liczba1 << endl;3

    liczba1 = 47;4
    cout << "Zmienna liczba1 ma teraz wartosc: " << liczba1 << endl;5

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_04

```
C:\biblioteczka\program_04\program_04.exe
Zmienna liczba1 ma wartosc: 21
Zmienna liczba1 ma teraz wartosc: 47
Aby kontynuować, naciśnij dowolny klawisz . . .
```

4 Od tej pory możemy już wykorzystywać zmienną w naszym kodzie (na przykład za pomocą polecenia `cout` 3 wyświetlić jej wartość na ekranie ●).

5 Oczywiście, zgodnie z ideą zmiennej, możemy w każdej chwili zmienić jej wartość. Wystarczy wpisać jej nazwę, znak równości i nową wartość 4. Jak można się tego spodziewać, jeśli teraz wyświetlimy zmienną 5, na ekranie pojawi się jej nowa wartość ●. Wartość zmiennej możemy zmieniać wielokrotnie wewnątrz kodu programu. Operację nadania lub zmiany wartości nazywamy przypisaniem. Pamiętajmy, że zmienna może przechowywać tylko wartości takiego typu, jaki ustaliliśmy podczas deklaracji zmiennej.

▼ Deklaracja, inicjacja i definicja

Ze stałymi i zmiennymi związane są trzy często występujące pojęcia: deklaracja, inicjacja i definicja. Pierwsze określa typ wartości, jakie może przechowywać dana stała lub zmienna (na przykład ●). Inicjacja to pierwsze przypisanie stałej lub zmiennej wartości ● i powoduje przydzielenie pamięci dla zmiennej. Definicja to jednocześnie deklaracja i inicjacja (przykładowy zapis definicji zmiennej mógłby wyglądać tak ●).

```
int liczba;
liczba = 21;
```

```
int liczba = 21;
```

Jak nazywać stałe i zmienne

Nazwa stałej czy zmiennej powinna jednocześnie dobrze określać jej znaczenie i spełniać szereg kryteriów. Oto najważniejsze z nich:

- Nazwa stałej lub zmiennej powinna być słowem lub skrótem, który określa jej przeznaczenie.
- Nazwa może się składać z małych i wielkich liter (wielkość liter jest rozróżniana przez kompilator), a także cyfr i znaku podkreślenia (_). Pamiętajmy jednak, że cyfra nie może rozpoczynać nazwy stałej lub zmiennej. Nie można także używać znaków narodowych (na przykład liter ź czy ą).
- Przyjęło się, że nazwy stałych pisane są wielkimi literami (na przykład PROMIEN), a nazwa zmiennej powinna rozpoczynać się małą literą (na przykład liczba).

Jak widzimy, nazwa stałej lub zmiennej może przybierać różne postacie. Należy jednak pamiętać, że zarówno stała, jak i zmienna nie mogą mieć nazw zarezerwowanych dla poleceń języka C. W tabeli Komputer ŚWIAT zestawił większość słów kluczowych, których nie można wykorzystać jako nazwy dla stałych i zmiennych.

auto	break	case
char	class	const
continue	default	do
double	else	enum
extern	float	for
goto	if	int
long	private	register
return	short	signed
sizeof	static	struct
switch	try	typedef
unsigned	void	while

Wczytywanie z klawiatury

Dzięki możliwości wczytywania danych z klawiatury użytkownik może się komunikować z programem

Wiemy już, czym są stałe i zmienne. Poznaliśmy sposób wyświetlania ich wartości na ekranie naszego komputera. Warto więc dowiedzieć się, w jaki sposób komunikować się z naszym programem, czyli w trakcie jego działania ustalać wartość zmiennej za pomocą naszej klawiatury.

1 Stwórzmy nowy projekt (nazwijmy go na przykład program_05).

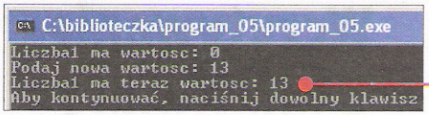
2 Wewnątrz funkcji `main` deklarujemy zmienną `liczba1` i inicjujemy ją wartością 0.

Program_05

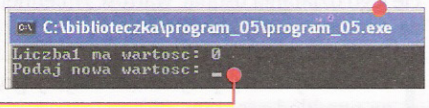
```
int liczba1 = 0;
cout << "Liczba1 ma wartosc: " << liczba1 << endl;

cout << "Podaj nowa wartosc: ";
cin >> liczba1;

cout << "Liczba1 ma teraz wartosc: " << liczba1 << endl;
```

3 Za pomocą strumienia `cout` sprawdźmy, czy `liczba1` rzeczywiście ma ustaloną przez nas wartość. Jeśli byśmy skompilowali i uruchomili w tej chwili nasz projekt, na ekranie zobaczylibyśmy napis `Liczba1 ma wartosc: 0`.

4 Wyświetlamy na ekranie informację, aby użytkownik wpisał nową wartość zmiennej `liczba1`, i wpisujemy instrukcję `cin` odpowiedzialną za pobranie z klawiatury liczby i przypisanie jej do zmiennej `liczba1`.

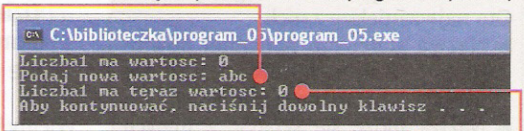


5 Aby sprawdzić, jaka wartość zostanie wpisana do zmiennej `liczba1`, wyświetlamy ją ponownie na ekranie. Po skompilowaniu i uruchomieniu naszego programu na ekranie zobaczymy widok ●. Miganie kursora ● oznacza czekanie programu na reakcję użytkownika. Wpisujemy dowolną liczbę i wciskamy klawisz enter.



▶ Komputer ŚWIAT radzi

Naszą zmienną `liczba1` zadeklarowaliśmy jako całkowitoliczbową. Sprawdźmy, co się stanie, jeśli podczas działania programu wpiszemy



na klawiaturze na przykład litery ●. Szybko się przekonamy, że `liczba1` nie zmieni swojej wartości ●.

6 Zostaje wykonana dalsza część programu – jak możemy się sami przekonać, zmienna `liczba1` ma ustaloną przez nas wartość ●.

▼ Jak to wygląda na schemacie

Wiemy już, jak w kodzie języka C++ wyświetlić i wczytać zmienną. Warto wiedzieć, w jaki sposób można zobrazować to na schemacie blokowym.

Wczytywanie i wyświetlanie danych (na przykład zmiennych) reprezentowane jest przez równoległobok, w którym znajduje się informacja o przeprowadzanej operacji i nazwie zmiennej, do której została zapisana wczytana z klawiatury wartość, lub nazwie zmiennej, która została skierowana na ekran. Inaczej mówiąc, każda operacja wejścia/wyjścia to na schemacie równoległobok. Spójrzmy na przykładowy algorytm i spróbujmy go przeanalizować.

1 Bloki ● już znamy – to oznaczenie początku i końca programu.

2 Figura ● to operacja wejścia/wyjścia. Znajdujący się wewnątrz symbolu napis informuje nas, że program pobiera od użytkownika wartość zmiennej `liczba`. Blok ● to ponownie operacja wejścia/wyjścia. Napis wewnątrz figury mówi, że na ekranie zostanie wyświetlona wartość pobranej wcześniej zmiennej `liczba`.

Jak działa pobieranie wartości z klawiatury

Instrukcja `cin` (czytamy jako si-in) jest przeciwieństwem polecenia `cout` i oznacza standardowy strumień wejściowy. Zwróćmy uwagę na symbol, który pojawia się bezpośrednio po `cin`. Są to, podobnie jak w wypadku standardowego strumienia wyjściowego, dwa nawiasy ostre, z tą różnicą, że skierowane w drugą stronę. Przy wyświetlaniu symbol ten wskazuje na instrukcję `cout`, w instrukcji pobierania wartości na nazwę zmiennej, która się za nim znajduje. Tak więc linię `cin >> liczba1;` należy odczytywać w taki sposób: ze standardowego strumienia wejściowego (domyślnie z klawiatury) pobierz wartość i zapisz ją w zmiennej `liczba1`.

Operacje na zmiennych

Wiemy już, co to są zmienne. Poznajmy teraz tajniki posługiwania się nimi do obliczeń matematycznych i logicznych. Wiedza ta będzie nam już niedługo bardzo potrzebna

Przedstawione do tej pory możliwości stałych i zmiennych nie pozwalają stworzyć nam żadnej użytecznej aplikacji. Po poznaniu pojęcia operatorów i wyrażeń sytuacja ta się na pewno zmieni.

Operatory a wyrażenia

Operatory umożliwiają zapisywanie wyrażeń. Prawie każdy operator oznaczany jest charakterystycznym dla niego symbolem. Najlepszym przykładem będzie z pewnością operator + (plus). Za jego pomocą możemy stworzyć na przykład wyrażenie 15+7, które polega na dodaniu do liczby 15 liczby 7.

Operatory matematyczne

symbol	znaczenie
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie
%	reszta z dzielenia

Czym jest wyrażenie

Wyrażenie jest kombinacją stałych, zmiennych i operatorów, które stosuje się najczęściej do zapisu operacji matematycznych. Warto pamiętać, że pojęcie stałej nie ogranicza się wyłącznie do nazwy poprzedzonej słowem kluczowym `const`. Jako stałą należy również traktować każdą liczbę, literę, a także ciąg znaków znajdujących się w cudzysłowach. Wynika to z faktu, że konkretnej liczby, litery czy ciągu znaków nie można w żaden sposób zmienić i przypisać im innej wartości (czyli można powiedzieć, że liczba 7 to stała, której wartość wynosi 7).

Operatory matematyczne

Najpopularniejszymi i niewymagającymi dokładnego opisu operatorami są operatory arytmetyczne. Pierwsze cztery nie wyma-

Program_06

```
int x;
int y;

cout << "Podaj dwie liczby:";
cin >> x >> y;

cout << "Suma: " << x + y << endl;
cout << "Roznica: " << x - y << endl;
cout << "Iloczyn: " << x * y << endl;
cout << "Iloraz: " << x / y << endl;
cout << "Reszta z dzielenia: " << x % y << endl;
```

1 Na początku w funkcji `main` deklarujemy dwie zmienne całkowitoliczbowe o nazwach `x` i `y`.

2 Wykorzystując strumień wejściowy `cin`, pobieramy te dwie liczby z klawiatury.

3 W następnych linijkach wyświetlamy wyniki kolejnych wyrażeń. Po skompilowaniu i uruchomieniu programu oraz wprowadzeniu wartości zmiennej `x` i `y` na ekranie widzimy na przykład.

```
C:\biblioteczka\program_06\program_06.exe
Podaj dwie liczby:16
6
Suma: 22
Roznica: 10
Iloczyn: 96
Iloraz: 2
Reszta z dzielenia: 4
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Komputer ŚWIAT radzi

Zwróćmy uwagę, że podobnie jak w wypadku strumienia wyjściowego `cout` możemy pobrać od razu dwie zmienne.

gają wyjaśnienia. Warto tylko pamiętać, że operatory mnożenia i dzielenia, tak jak w matematyce, mają większy priorytet. Czyli wyrażenie 7+4*3 da w rezultacie wynik 19, a nie 33. Oczywiście możemy zmienić sposób obliczania tego wyrażenia, stosując nawiasy (wyłącznie nawiasy okrągłe). Na przykład wynikiem wyrażenia (7+4)*3 będzie 33.

Wyjaśnienia wymaga natomiast symbol %. Zwraca on resztę z dzielenia liczby znajdującej się przed tym symbolem przez liczbę znajdującą się za nim. Przykładowo wyrażenie 13%3 da w rezultacie 1, ponieważ liczba 3 mieści się w liczbie 13 cztery razy i pozostaje jeszcze reszta 1. Symbol % bywa też często nazywany dzieleniem modulo.

Wszystkie operatory matematyczne są dwuargumentowe. Oznacza to, że z ich lewej i prawej strony muszą znajdować się wartości (oczywiście wartości te mogą być zapisane w zmiennej czy stałej).

Aby przetestować działanie operatorów matematycznych, stwórzmy w Dev C++ nowy projekt (nazwijmy go na przykład program_06).

Operator przypisania

Wiemy już, że skierowanie do strumienia wyjściowego `cout` wyrażenia arytmetycznego powoduje wyświetlenie na ekranie wyniku tego wyrażenia. Jednak stosowanie wyrażeń nie ogranicza się wyłącznie do obliczania i wyświetlania prostych operacji matematycznych. Wynik dowolnego wyrażenia możemy również przypisać do zmiennej. Do tego celu służy operator przypisania reprezentowany przez symbol =. Wykorzystywaliśmy go już do zainicjowania stałej i zmiennej. Zobaczmy, w jaki sposób

Komputer ŚWIAT radzi

Jeśli w wyrażeniu wystąpią tylko liczby całkowite (a takie są przechowywane w zmiennych `x` i `y` zadeklarowanych jako `int`), wynik wyrażenia będzie również całkowity.

można jeszcze wykorzystać operator przypisania.

Program_07a

```
int x, y, z;

cout << "Podaj dwie liczby:";
cin >> x >> y;

z = 2 * (x + 7 * y) / 3;

cout << "Zmienna z wynosi: " << z << endl;
```

1 Po stworzeniu nowego projektu w funkcji `main` deklarujemy istnienie trzech zmiennych (demonstrując przy okazji działanie operatora łączenia – więcej informacji w ramce poniżej).

Operator łączenia

Dzięki operatorowi łączenia (reprezentuje go symbol przecinka) możemy w jednej linijce zadeklarować od razu kilka zmiennych. W tym wypadku wszystkie zmienne będą typu całkowitoliczbowego (`int`).

2 Prosimy użytkownika o wpisanie wartości dwóch zmiennych (`x` i `y`).

3 Do zmiennej `z` przypisujemy (za pomocą symbolu równości, czyli operatora przypisania) wynik wyrażenia.

```
c:\biblioteczka\program_07a\program_07a.exe
Podaj dwie liczby:11
4
Zmienna z wynosi: 26
Aby kontynuować, naciśnij dowolny klawisz . . .
```

4 Aby sprawdzić, jaka wartość znajduje się w zmiennej `z`, wyświetlamy ją na ekranie. Po uruchomieniu programu i wprowadzeniu dwóch zmiennych możemy się rzeczywiście przekonać, że do zmiennej `z` przypisany został wynik działania wyrażenia.

Zwiększanie i zmniejszanie o 1

W języku C++ występują dwa szczególne operatory – inkrementacji (zwiększania o 1) i dekrementacji (zmniejszania o 1). Symbolem pierwszego z nich są dwa plusy ++, a drugiego dwa minusy --. Warto poznać właściwości obu operatorów oraz sposób ich stosowania. Z dalszej lektury tej książki dowiemy się, że są one niezwykle często wykorzystywane.

Zobaczmy teraz, jak posługiwać się operatorami inkrementacji i dekrementacji.

1 Na początku deklarujemy zmienną `x`, wyświetlamy na ekranie prośbę o podanie jej wartości i pobieramy wpisaną przez użytkownika liczbę (1).

2 Aby się przekonać, ile rzeczywiście wynosi zmienna `x`, wyświetlamy ją na ekranie.

Dwa przypisania naraz

Jeśli chcielibyśmy dwóm zmiennym nadać tę samą wartość, możemy to zrobić za pomocą jednej instrukcji (czyli w jednej linii kodu). Na przykład zapis `x = y = 10;` spowoduje przypisanie zmiennej `x` i zmiennej `y` wartości 10. Warto wiedzieć, że wykonanie takiej instrukcji rozpocznie się od przypisania zmiennej `y` wartości 10, a następnie zmiennej `x` aktualnej wartości, którą ma zmienna `y` (czyli 10).

3 Następnie inkrementujemy zmienną `x` ❷, czyli wykonujemy instrukcję zwiększenia jej wartości o 1 przy wykorzystaniu operatora ++.

Program_07b

```
int x;
cout << "Podaj dowolna liczbe: ";
cin >> x; ❶
cout << "Liczba wynosi :" << x << endl;
x++; ❷
cout << "Liczba wynosi :" << x << endl;
x--; ❸
cout << "Liczba wynosi :" << x << endl;
```

4 Aby sprawdzić, jak zmieniła się wartość zmiennej `x`, wyświetlamy ją na ekranie. Jak możemy się łatwo przekonać, zmienna zwiększyła się o 1 ●.

5 Sprawdźmy jeszcze, co się stanie po zastosowaniu operatora dekrementacji ❸. Okaże się, że zmienna zmniejszyła się o 1 i wynosi tyle, ile na początku ●.

►Komputer ŚWIAT radzi

Jeśli przypomnimy sobie porównanie zmiennej do pieczątki z możliwością zmiany, to inkrementację i dekrementację możemy porównać do przekręcenia o jedną pozycję do przodu lub do tyłu kółeczka zmieniającego wartość pieczątki.

Przedrostkowa i przyrostkowa inkrementacja i dekrementacja

Przedstawione przed chwilą przez Komputer ŚWIAT operatory inkrementacji i dekrementacji występowały w tak zwanej przyrostkowej wersji. Istnieje jednak jeszcze drugi sposób zapisu zwiększania i zmniejszania o 1 – tak zwany przedrostkowy. Polega on na tym, że operator ++ lub -- występuje przed nazwą zmiennej. Jego działanie się oczywiście nie zmienia (nadal zwiększa lub zmniejsza wartość zmiennej), jednak zasadniczo zmienia się sposób wykonywania tej modyfikacji. Jeśli w przedstawionym przed chwilą przykładzie zamienilibyśmy wersję przyrostkową na przedrostkową, efekt działania programu byłby identyczny. Sprawdźmy więc, w jakim przypadku działanie programu będzie inne i na czym polega różnica pomiędzy tymi dwoma sposobami zapisu inkrementacji i dekrementacji.

1 Zmodyfikujmy nieco nasz ostatni program. Zamiast wykonywać instrukcję inkrementacji i dopiero potem wyświetlać zmienną, do standardowego strumienia wyjściowego przekażemy od razu wyrażenie `x++` i dla porównania wyrażenie `++x`. Na początek, tak jak poprzednio, pobieramy wartość zmiennej `x`.

```
c:\biblioteczka\program_07b\program_07b.exe
Podaj dowolna liczbe: 13
Liczba wynosi :13
Liczba wynosi :14 ●
Liczba wynosi :13 ●
Aby kontynuować, naciśnij dowolny klawisz . .
```

2 Następnie do strumienia `cout` skierujmy wyrażenie `x++` ●. Co się wyświetli na ekranie? Wbrew pozorom, wyświetlona na ekranie wartość nie będzie

Program_08

```
int x;
cout << "Podaj dowolna liczbe: ";
cin >> x;

cout << "Liczba wynosi :" << x++ << endl;
cout << "Liczba wynosi :" << x << endl;
cout << endl;
cout << "Liczba wynosi :" << ++x << endl;
cout << "Liczba wynosi :" << x << endl;
```

zwiększona o 1 ●. Dlaczego tak się stało? Ponieważ do strumienia `cout` najpierw została przekazana zmienna `x` (z wpisaną przez nas z klawiatury wartością), a dopiero później zadziałał operator inkrementacji, zwiększając wartość zmiennej o 1. Ale czy rzeczywiście zadziałał? Przekonamy się o tym, ponownie wyświetlając wartość zmiennej. Na ekranie pojawi się zwiększona o 1 wartość ●.

3 Dla porównania sprawdźmy, co się stanie, jeśli zastosujemy wspomnianą już przedrostkową wersję zapisu operatora inkrementacji. Na ekranie pojawia się od razu zwiększona o 1 wartość zmiennej `x` ●. Dzieje się tak dlatego, że zmienna najpierw została poddana inkrementacji i do strumienia `cout` trafiła już zmieniona wartość zmiennej.

```
c:\biblioteczka\program_08\program_08.exe
Podaj dowolna liczbe: 20
Liczba wynosi :20
Liczba wynosi :21

Liczba wynosi :22
Liczba wynosi :22
Aby kontynuować, naciśnij dowolny klawisz
```

▶ Komputer ŚWIAT radzi

Warto bliżej przyjrzeć się przykładowi przypisania z mnożeniem z tabeli poniżej. Zauważmy, że zapis `x *= y - 2` równoważny jest zapisowi `x = x * (y - 2)`, a nie zapisowi `x = x * y - 2`, gdzie najpierw wykonywane jest mnożenie, a później odejmowanie.

4 Aby się upewnić, że wyświetlona wartość zmiennej jest ostateczna, pokażmy ją jeszcze raz na ekranie.

Arytmetyczne operatory przypisania

Twórcy języka C++ zadbali o to, aby ułatwić życie programiście i uprościć pracę kompilatora. Najlepszym tego przykładem są arytmetyczne operatory przypisania, czyli takie konstrukcje języka, które pozwalają za pomocą jednego symbolu zmienić wartość i przypisać ją na przykład do zmiennej. Jak zobaczymy z dalszej lektury, wykorzystanie arytmetycznych operatorów przypisania ma ogromne znaczenie w wielu innych konstrukcjach językowych. Spójrzmy na przykład, którego kod znajduje się na następnej stronie.

1 Załóżmy, że pobieramy od użytkownika zmienną `x` i naszym zadaniem jest dodanie do `x` liczby 20. Jak to zrobić? Moglibyśmy sumę `x` i 20 przypisać innej zmiennej, a następnie wartość tej zmiennej przypisać do `x`. Jest jednak znacznie prostsze

▼ Arytmetyczne operatory przypisania

operator	nazwa	przykład	równoważny zapis
+=	przypisanie z dodawaniem	x += 11	x= x + 11
-=	przypisanie z odejmowaniem	y -= x	y = y – x
*=	przypisanie z mnożeniem	x *= y – 2	x = x * (y – 2)
/=	przypisanie z dzieleniem	z /= x	z = z / x
%=	przypisanie reszty dzielenia	x %= 7	x = x % 7

rozwiązanie – możemy to zrobić na przykład tak.

Program_09

```
int x;
cout << "Podaj x: ";
cin >> x;
cout << "Zmienna x wynosi: " << x << endl;

x = x + 20;
cout << "Teraz zmienna wynosi: " << x << endl;

x += 20;
cout << "Teraz zmienna wynosi: " << x << endl;
```

```
c:\biblioteczka\program_09\program_09.exe
Podaj x: 30
Zmienna x wynosi: 30
Teraz zmienna wynosi: 50
Teraz zmienna wynosi: 70
```

2 Aby się przekonać, że wartość `x` uległa zmianie, wyświetlamy ją na ekranie. Rzeczywiście, `x` zostało powiększone o 20. Działanie zapisu `x = x + 20` polega na dodaniu do starej wartości zmiennej `x` liczby 20, a następnie przypisaniu tej sumy do zmiennej `x`.

3 Zapis `x = x + 20` można znacznie uprościć i zapisać w taki sposób. Symbol `+=` to właśnie jeden z arytmetycznych operatorów przypisania. Jak się przekonamy po kompilacji i uruchomieniu naszego programu, działanie arytmetycznego operatora przypisania jest identyczne jak działanie zapisu `x = x + 20`.

Przypisanie ze zmianą wartości o 1

Zauważmy, że poznany wcześniej zapis `x++` lub `++x` powoduje taki sam efekt, jak zapis z wykorzystaniem arytmetycznego operatora przypisania `x += 1`. Tak więc do zapisania, zwiększenia lub zmniejszenia zmiennej o 1 można skorzystać z bardzo wielu możliwości – wynika to z tego, że inkrementacja i dekrementacja to bardzo często występująca w C++ (a także w każdym innym języku programowania) konstrukcja. Warto więc wiedzieć, w jaki sposób kompilator zachowa się w wypadku na przykład takich zapisów.

1 W wypadku instrukcji na początku zostaje wykonane przypisanie wartości zmiennej `x` do zmiennej `y`, a dopiero po tym przypisaniu zmienna `x` zostanie zwiększana o 1. Tak więc wynikiem takiej instrukcji będzie w tym wypadku wartość 1 zapisana w zmiennej `y` i wartość 2 zapisana w zmiennej `x`.

2 Instrukcja zostanie wykonana w trochę inny sposób. Najpierw zostanie zwiększona zmienna `x` o 1, a następnie zmieniona już wartość zostanie przypisana do zmiennej `y`. Czyli po uruchomieniu programu obie zmienne będą miały tę samą wartość równą 2.

```
c:\bi
x = 2
y = 1

x = 2
y = 2
```

Program_10

```
int x, y;

x = y = 1;
y = x++;
cout << "x = " << x << endl << "y = " << y << endl << endl;

x = y = 1;
y = ++x;
cout << "x = " << x << endl << "y = " << y << endl;
```

Odejmowanie a znak minusa

Opisany już przez Komputer ŚWIAT operator odejmowania ma też swoje drugie znaczenie. Może on oznaczać liczbę ujemną. Przykładem może być zapis ●. Aby uniknąć niejednoznaczności (w czytaniu kodu, bo kompilator sobie z nim bez problemu poradzi), lepiej znak ujemności zapisywać w nawiasach okrągłych ●.

```
x = 100 - -x;
x = 100 - (-x);
```

Relacje między zmiennymi

Wiemy już, jak zmienić wartość zmiennej, przypisać ją do innej zmiennej czy wykonać operacje arytmetyczne. Ale jak sprawdzić, czy wartość zmiennej jest większa niż inna liczba lub wartość innej zmiennej? Służą do tego specjalne operatory zwane operatorami relacji. Choć są bardzo ważne, nie będziemy w tym miejscu poświęcali im zbyt dużo miejsca. Przestudiujmy jedynie tabelkę Operatory relacji i spójrzmy na przykład ●.

Program_11

```
int x, y, z;
x = 5;
y = 10;
z = 15;

cout << "wynikiem relacji x < y jest " << (x < y) << endl;
cout << "wynikiem relacji x > y jest " << (x > y) << endl;

cout << "wynikiem relacji x == y jest " << (x==y) << endl;
cout << "wynikiem relacji z == 15 jest " << (z==15) << endl;
```

```
c:\biblioteczka\program_11\program_11.ex
wynikiem relacji x < y jest 1
wynikiem relacji x > y jest 0
wynikiem relacji x == y jest 0
wynikiem relacji z == 15 jest 1
```

1 Na początku deklarujemy trzy zmienne i przypisujemy im kolejno wartości 5, 10 i 15.

2 Następnie wyświetlamy na ekranie kilka przykładowych relacji ●. Po skompilowaniu i uruchomieniu naszego programu łatwo możemy zobaczyć, że rezultatem prawdziwej relacji jest liczba 1 (na przykład ●). Nieprawdziwą (inaczej mówiąc fałszywą) relację reprezentuje liczba 0 (na przykład ●). W ramce Prawda czy fałsz przeczytamy o często używanych pojęciach określających wynik relacji.

Prawda czy fałsz

Zamiast liczb 1 i 0 możemy operować pojęciami prawda i fałsz. W programowaniu to bardzo częsta interpretacja. Warto wiedzieć, że system dwójkowy przedstawia się również jako „jest sygnał" i „brak sygnału".

Operatory relacji

operator	nazwa	przykład	wynik
==	równe	12 == (10 + 2)	1 (prawda)
!=	różne od	7!= (9 – 2)	0 (fałsz)
<	mniejsze	3 < 9	1 (prawda)
>	większe	4 > (2 + 7)	0 (fałsz)
<=	mniejsze lub równe	5 <= (2 + 3)	1 (prawda)
>=	większe lub równe	4 >= 5	0 (fałsz)

Nasz pierwszy przydatny program

Teoria to nie wszystko. Sprawdźmy więc w praktyce zdobytą do tej pory wiedzę z zakresu programowania

Stworzyliśmy już kilka drobnych programów, które nie realizowały zbyt wiele, ale dobrze pokazywały działanie omawianych konstrukcji języka C++. Przyszedł jednak czas na napisanie pożytecznej aplikacji. Komputer ŚWIAT pokaże, jak napisać program obliczający pola powierzchni różnych figur płaskich. Na początku, przed stworzeniem właściwego kodu programu, zgodnie z opisywanymi wcześniej zaleceniami, zaprojektujemy naszą aplikację i stworzymy schemat blokowy prezentujący jej działanie.

Określamy cel

Chcemy napisać program, więc musimy wiedzieć, co będzie robił. Określamy w punktach jego założenia:

- program będzie obliczał pola powierzchni figur: prostokąta, trójkąta i koła,
- wszystkie niezbędne do obliczenia dane będzie można wprowadzić z klawiatury,
- kolejne pola będą obliczane po kolei,
- na końcu porównamy obliczone pola figur i wyświetlimy różne relacje.

Piszemy algorytm

Wiemy, co będzie robił program. Postawmy więc sobie pytanie, jak będzie to robił. Napiszmy jego algorytm – najlepiej w punk-

```
const int PI = 3;
int a, b, x, h, r, pole_prostokata, pole_trojkata, pole_kola;

cout << "Podaj dlugosc bokow prostokata: ";
cin >> a >> b;
pole_prostokata = a * b;
cout << "Pole prostokata o bokach dlugosci " << a << " i " << b;
cout << " wynosi " << pole_prostokata << endl << endl;

cout << "Podaj dlugosc podstawy oraz wysokosc trojkata: ";
cin >> x >> h;
pole_trojkata = x * h / 2;
cout << "Pole trojkata, w ktorym podstawa ma " << x << " a wysokosc " ;
cout << h << " wynosi " << pole_trojkata << endl << endl;

cout << "Podaj promien kola: ";
cin >> r;
pole_kola = PI * r * r;
cout << "Pole kola o promieniu " << r << " wynosi " << pole_kola;
cout << endl << endl;

cout << "Pole prostokata jest wieksze od pola kola? Odpowiedz: ";
cout << (pole_prostokata > pole_kola) << endl;

cout << "Pole kola jest rozne od pola trojkata? Odpowiedz: ";
cout << (pole_kola != pole_trojkata) << endl;
```

Program_12

tach. A więc w opisowym języku polecenia dla komputera mogłyby wyglądać tak:

- pobierz dane o długości boków prostokąta i zapisz je w zmiennych `a` i `b`,
- oblicz pole powierzchni prostokąta, zapisz w zmiennej `pole_prostokata` i wyświetl wynik na ekranie,
- pobierz długości podstawy i wysokości trójkąta i zapisz je w zmiennych `x` i `h`,
- oblicz pole powierzchni trójkąta, zapisz w zmiennej `pole_trojkata` i wyświetl wynik na ekranie,
- pobierz długość promienia koła i zapisz w zmiennej `r`,
- oblicz pole koła, zapisz w zmiennej `pole_kola` i wyświetl wynik,
- porównaj obliczone pola i wyświetl wyniki relacji w postaci zer i jedynek.

► Komputer ŚWIAT radzi

Wszystkie operacje na zmiennych (obliczenia, przypisania, inkrementacje i dekrementacje) na schemacie blokowym przedstawiamy za pomocą prostokąta.

Schemat blokowy

Wymyślony przez nas algorytm działania jest na tyle prosty, że tworzenie schematu blokowego wydaje się bezcelowe. Przy większych programach nie będzie to oczywiste. Jeśli nabierzemy wprawy, zrezygnujmy z rozpisywania algorytmów. Warto jednak zawsze stworzyć dobry schemat blokowy. Dla naszego programu będzie on wyglądał tak ●. Komputer ŚWIAT poleca przeanalizowanie schematu oraz spróbowanie jego ulepszenia.

Schemat blokowy programu

```
Podaj dlugosc bokow prostokata: 4
5
Pole prostokata o bokach dlugosci 4 i 5 wynosi 20

Podaj dlugosc podstawy oraz wysokosc trojkata: 10
4
Pole trojkata, w ktorym podstawa ma 10 a wysokosc 4 wynosi 20

Podaj promien kola: 4
Pole kola o promieniu 4 wynosi 48

Pole prostokata jest wieksze od pola kola? Odpowiedz: 0
Pole kola jest rozne od pola trojkata? Odpowiedz: 1
```

Piszemy aplikację

Po wyznaczeniu celów naszego programu, rozpisaniu jego poszczególnych działań oraz narysowaniu schematu blokowego zakodowanie programu w C++ wydaje się już tylko formalnością – kod znajdziemy na stronie 46. Po jego skompilowaniu i uruchomieniu na ekranie zobaczymy taki widok.

► Komputer ŚWIAT radzi

Oczywiście algorytm, schemat blokowy, a wreszcie kod realizujący postawione przez nas zadanie może wyglądać inaczej. Wszystko zależy od doświadczenia programisty. Warto wiedzieć, że im lepiej rozpisany algorytm i schemat blokowy, tym kod będzie na pewno czytelniejszy.

Typy danych

Każda zmienna może przechowywać tylko określone dane. Zobaczmy, jak ustalać typ danych

Opanowaliśmy już umiejętność posługiwania się zmiennymi. Dowiedzieliśmy się też, że każda zmienna może się zmieniać tylko w określonym zakresie – tak jak na dającej się przestawiać pieczątce z datą nie ustawimy dowolnego napisu. Do tej pory operowaliśmy zmiennymi całkowitoliczbowymi (zapewniało nam to słowo kluczowe `int` przed nazwą zmiennej w jej deklaracji). Na pewno zauważyliśmy, że zmienne całkowitoliczbowe mają spore ograniczenia i dokładne obliczenie (na przykład pola koła) z ich wykorzystaniem nie są możliwe. Czas poznać inne typy zmiennych oraz ich różne modyfikacje.

Podział typów

Występujące w C++ typy danych można podzielić na trzy rodzaje:

- arytmetyczne,
- logiczne,
- inne.

Na rysunku Komputer ŚWIAT schematycznie pokazał podział typów danych. Możemy zauważyć, że do typów arytmetycznych należą typy całkowitoliczbowe, zmiennopozycyjne oraz znakowe. Wśród innych typów możemy wyróżnić typ nieokreślony, wskaźnikowy, referencyjny, wyliczeniowy, tablicowy. Przyjrzyjmy się poszczególnym typom bliżej – zaczniemy od arytmetycznych.

Liczby całkowite

Wiemy już, że wpisując w deklaracji zmiennej przed jej nazwą słowo kluczowe `int`, ustalamy, że zmienna ta będzie typu całkowitoliczbowego, czyli będzie mogła przechowywać liczby całkowite. Jest to jednak duże uproszczenie. Tak na prawdę zmienna zadeklarowana jako `int` może przechowywać liczby całkowite ze znakiem (czyli liczby ujemne i dodatnie) od –2147483648 do 2147483647. Jeśli chcemy się o tym przekonać, wpiszmy kod. Jego działanie jest proste – program pobiera z klawiatury wartość i zapisuje ją w zmiennej `liczba`. Po uruchomieniu programu wpiszmy liczbę należącą do danego zakresu (na przykład). Zostanie ona poprawnie przypisana do zmiennej. Spróbujmy teraz wpisać liczbę większą lub mniejszą niż granice typu zmiennej `liczba` (na przykład). Wpisana przez nas liczba nie zostanie poprawnie zapisana w zmiennej.

Program_13

```
int liczba;
cout << "Podaj liczbe: ";
cin >> liczba;
```

```
c:\biblioteczka\program_13\program_13.exe
Podaj liczbe: -2147483647
Liczba wynosi: -2147483647
```

```
c:\biblioteczka\program_13\program_13.exe
Podaj liczbe: 2147483650
Liczba wynosi: 2
```

Typy wbudowane

Wszystkie wymienione typy danych są wbudowane w język C++. Oznacza to, że określone są przez twórców tego języka. Oprócz wbudowanych typów danych można tworzyć swoje własne typy danych. Przykładem może być typ `string` (poznaliśmy go przy okazji omawiania stałych). Zdefiniowany on jest w jednej z bibliotek i mogliśmy z niego korzystać tylko dzięki temu, że dołączyliśmy tę bibliotekę do naszego kodu za pomocą dwóch dyrektyw `#include`, które są automatycznie umieszczane przez aplikację Dev-C++ w szkielecie programu.

Komputer ŚWIAT radzi

Standard języka C++ nie definiuje zakresów wartości, jakie mogą przechowywać zmienne danego typu. Podany przez Komputer ŚWIAT zakres dla typu `int` jest prawidłowy dla większości systemów 32-bitowych, jednak w wypadku innych systemów i innych kompilatorów może być on inny. Podany przez Komputer ŚWIAT zakres wynika z tego, że w pamięci na zmienną typu `int` przeznaczone są 4 bajty (czyli 32 bity), a więc można w niej zapisać 2^{32} różnych liczb.

Typy całkowitoliczbowe

definicja	zakres	
	od	do
unsigned short int	0	65535 (2^{16}-1)
signed short int	-32768 (-2^{15})	32767 (2^{15}-1)
short int	-32767 (-2^{15}+1)	32768 (2^{15})
unsigned int	0	4294967295 (2^{32}-1)
signed int	-2147483647 (-2^{31}+1)	2147483648 (2^{31})
int	-2147483648 (-2^{31})	2147483647 (2^{31}-1)
unsigned long int	0	4294967295 (2^{32}-1)
signed long int	-2147483647 (-2^{31}+1)	2147483648 (2^{31})
long int	-2147483648 (-2^{31})	2147483647 (2^{31}-1)

ŚWIAT przedstawił wszystkie możliwe kombinacje modyfikatorów typu `int`. W tabeli znajdziemy również najczęściej występujące zakresy liczb, które można przechowywać w zmiennej określonego typu całkowitoliczbowego.

Wiemy już, jakie dane można umieścić w zmiennej zdefiniowanej za pomocą słowa kluczowego `int`. Twórcy języka pozostawili jednak swobodę programistom i dzięki tak zwanym modyfikatorom możemy nieco wpływać na zakres danych, które można zapisać w zmiennej `int` (oraz, jak się później dowiemy, również w zmiennych innych typów). Modyfikatory to nic innego jak określenia różnych odmian jednego typu. Oto najważniejsze z modyfikatorów:

- `short` – ang. krótki – pozwala na zmniejszenie zakresu liczb, a dzięki temu na zmniejszenie ilości pamięci wykorzystywanej do przechowywania zmiennej,
- `long` – ang. długi – dzięki niemu zwiększymy zakres liczb, które mogą być przechowywane w zmiennej,
- `signed` – określa, że dany typ reprezentuje dane dodatnie i ujemne (ze znakiem -),
- `unsigned` – określa, że liczby przechowywane w zmiennej tego typu mogą być tylko dodatnie.

Jak korzystać z modyfikatora? Wystarczy napisać go przed słowem kluczowym `int`. W tabelce Typy całkowitoliczbowe Komputer

Liczby z przecinkiem

Liczby całkowite nie pozwalają na przeprowadzanie dokładnych obliczeń (na przykład wspomnianego już wcześniej pola powierzchni koła). Do tego celu służą liczby zmiennopozycyjne (zwane również zmiennoprzecinkowymi), czyli takie, które pozwalają na zapisanie części ułamkowej. W języku C++ istnieją trzy typy, które pozwalają na zapisanie takich właśnie liczb:

- `float` – liczby zmiennopozycyjne z pojedynczą precyzją,
- `double` – ang. podwójne – liczby zmiennopozycyjne z podwójną precyzją,
- `long double` – liczby zmiennopozycyjne z rozszerzoną precyzją.

Komputer ŚWIAT radzi

Jak łatwo można zauważyć w tabelce, dopisanie modyfikatora `long` nie wpływa na rozszerzenie zakresu liczb. Jednak na starszych pecetach i kompilatorach, typ krótki i zwykły mają mniejszy zakres (na przykład ten pierwszy od -127 do 128, a drugi od -32767 do 32768), więc typ długi pozwala rzeczywiście na zapisanie większej liczby.

W tabeli Typy zmiennopozycyjne Komputer ŚWIAT przedstawił najczęściej występujące zakresy liczbowe.

Typy zmiennopozycyjne

definicja	zakres	
	od	do
float	$3{,}4 * 10^{-38}$	$3{,}4 * 10^{38}$
double	$1{,}7 * 10^{-308}$	$1{,}7 * 10^{308}$
long double	$3{,}4 * 10^{-4932}$	$3{,}4 * 10^{4932}$

Wiemy już, że typ zmiennoprzecinkowy pozwala na dokładne obliczenia. Aby naocznie się o tym przekonać, napiszmy kod będący rozszerzeniem fragmentu poprzednio stworzonego przez nas programu. Jego zadaniem będzie dokładne obliczenie pola koła (wynik działania poprzedniego kodu był zawsze liczbą całkowitą).

1 Na początku możemy zdefiniować jako stałą typu `float` liczbę pi. Dzięki temu, będzie ona bliższa jej prawdziwej wartości niż zapisana jako całkowita liczba 3.

2 Zmienne o nazwach `r` i `pole_kola` deklarujemy jako zmienne typu zmiennoprzecinkowego. Dzięki temu będziemy mogli podać dokładny promień koła, a obliczony przez program wynik będzie bardzo precyzyjny.

```
c:\biblioteczka\program_15\program_15.exe
Podaj promien kola: 3.6
Pole kola o promieniu 3.6 wynosi 40.715
```

3 Reszta programu jest taka sama, jak w wypadku kodu ze strony 46.

Program_15

```
const float PI = 3.141592;
float r, pole_kola;

cout << "Podaj promien kola: ";
cin >> r;
pole_kola = PI * r * r;
cout << "Pole kola o promieniu " << r << " wynosi " << pole_kola;
```

Sposób zapisu zmiennoprzecinkowych

Liczby zmiennoprzecinkowe można zapisywać na dwa sposoby. Pierwszy sposób, znany chyba wszystkim z matematyki, polega na zapisaniu części całkowitej liczby i po kropce jej części ułamkowej (na przykład). Pamiętajmy w tym wypadku, że część całkowitą od ułamkowej oddzielamy kropką, a nie przecinkiem.

Drugim sposobem zapisu, dużo wygodniejszym w wypadku bardzo małych lub bardzo dużych liczb, jest sposób wykładniczy. Jak go rozszyfrować? Wystarczy obliczyć przykładowe wyrażenie:

`2.7e4` $= 2{,}7 * 10^4 = 2{,}7 * 10\,000 = 27\,000$

O tym, czy tak jest rzeczywiście, przekonamy się, kompilując kod.

```
c:\biblioteczka\program_14\program_14.ex
34.768
27000
```

Program_14

```
float liczba1 = 34.768;
float liczba2 = 2.7e4;

cout << liczba1 << endl << liczba2 << endl << endl;
```

Komputer ŚWIAT radzi

Do stałej `PI` typu `float` Komputer ŚWIAT przypisał liczbę 3,141592. Zapis ten składa się z 7 cyfr znaczących. Niestety w zmiennej typu `float` można zapisać jedynie 6 cyfr znaczących. Jeśli więc wyświetlimy na ekranie stałą `PI`, okaże się, że została do niej przypisana liczba 3,14159 (bez ostatniej cyfry 2). Jeśli więc potrzebujemy naprawdę dokładnych obliczeń, korzystajmy z typu `double` (lub w ostateczności z typu `long double`).

Typ znakowy

Typ znakowy służy do reprezentacji pojedynczych znaków ASCII, czyli liter, cyfr, znaków przestankowych i innych specjalnych znaków widocznych na naszej klawiaturze (oraz wielu innych, których nie znajdziemy na klawiszach). Aby stworzyć zmienną typu znakowego, używamy słowa kluczowego `char`. Spójrzmy na przykład. Do zmiennej o nazwie `znak1` przypisujemy małą literę k – pamiętajmy, aby ująć ją w znaki apostrofu. Po uruchomieniu programu nasza litera zostanie wyświetlona na ekranie. Z ramki obok dowiemy się, że do zmiennej znakowej można przypisać również liczbę.

Program_16

```
char znak1 = 'k';
char znak2 = 107;

cout << znak1 << endl;
cout << znak2 << endl << endl;
```

```
c:\biblioteczka\program_16\program_16.e
k
k

Aby kontynuować, naciśnij dowolny kl
```

Prawda czy fałsz

Najprostszym, ale bardzo często używanym typem danych jest tryb logiczny. Do jego określenia służy słowo kluczowe `bool`. Do zmiennej typu logicznego możemy zapisać wyłącznie 0 lub 1. Zmienną tego typu najczęściej stosuje się do określenia, czy coś jest włączone, czy wyłączone lub prawdziwe albo fałszywe. Właśnie w takim znaczeniu poznaliśmy już typ logiczny – wartość `bool` zwracana była w wyniku operacji relacji (czyli porównania), na przykład w przykładzie na stronie 45. Jeśli relacja taka była prawdziwa (na przykład 7 > 3), to rezultatem była 1. Gdy relacja była nieprawdziwa, otrzymywaliśmy wartość 0.

Aby zobaczyć, w jaki sposób zdefiniować zmienną logiczną i jak z niej korzystać, przyjrzyjmy się przykładowi, którego efektem działania jest widok.

```
c:\biblioteczka\program_17\program_17.exe
1
0
```

Komputer ŚWIAT radzi

Wielu Czytelników na pewno się zastanawia, czemu typ znakowy zaliczamy do typów arytmetycznych, czyli liczbowych. Jednak tak naprawdę to typ znakowy jest typem liczbowym. Zmienna typu znakowego może bowiem przechować liczbę od 0 do 255, a każdemu znakowi w kodzie ASCII odpowiada konkretna liczba (na przykład małej literze k odpowiada liczba 107). Możemy się o tym przekonać, zapisując do zmiennej znakowej dowolną liczbę z przedziału od 0 do 255 – na przykład. Po wyświetleniu jej na ekranie zobaczymy zamiast liczby odpowiadający jej znak (w tym wypadku literę k).

Prawda, czyli true

Aby wyróżnić wartości zapisywane do zmiennej typu logicznego od zmiennej typu arytmetycznego, zamiast 1 i 0 stosuje się określenia `true` i `false`. Pierwsze z nich oznacza prawdę i przypisuje zmiennej wartość 1, drugie – fałsz i przypisuje zmiennej wartość 0.

Program_17

```
bool wlaczony;

wlaczony = 1;
cout << wlaczony << endl;

wlaczony = false;
cout << wlaczony << endl;
```

Inne typy

Znamy już wszystkie najważniejsze typy danych. W dalszej części książki poznamy jeszcze:

- typ void (w rozdziale V dotyczącym funkcji),
- typ wskaźnikowy pozwalający na tworzenie wskaźników do zmiennych i funkcji (więcej dowiemy się z rozdziału VI),
- typ tablicowy – pozwala na zapisywanie wielu liczb w jednej zmiennej – więcej w rozdziale VI,
- typ wyliczeniowy.

Pamięciożerne typy

Poznaliśmy już większość wbudowanych w C++ typów danych. Wiele z nich ma podobne zastosowanie – szczególnie wszystkie typu zmiennopozycyjne i całkowitoliczbowe służące głównie do przeprowadzania różnych obliczeń. Ale jaki typ stosować w danej sytuacji? Trzeba się nad każdym przypadkiem dobrze zastanowić.

Przy wyborze typu danych warto uwzględnić kilka ważnych czynników. Przede wszystkim typ deklarowany dla danej zmiennej powinien pozwolić jej przechowywać wartości pozwalające na dokonanie obliczeń z satysfakcjonującą nas precyzją. Oczywiście nie chodzi o wybranie za każdym razem typu zmiennopozycyjnego z podwójną precyzją, bo rzadko skorzystamy z zalet tak dokładnej liczby. Musimy uwzględnić też fakt zajmowanej przez zmienną pamięci. Im dokładniejszy typ danych – tym więcej miejsca zmienna tego typu zajmuje w pamięci naszego komputera. Jak sprawdzić, ile dokładnie? Można posłużyć się prostym kodem wykorzystującym specjalnie do tego celu stworzoną funkcję `sizeof`. Jej działanie polega na zwróceniu liczby określającej ilość bajtów potrzebnych do przechowania zmiennej danego typu. Po skompilowaniu i uruchomieniu kodu na ekranie zobaczymy widok. Łatwo się więc przekonamy, że zmienna typu `long double` zajmuje w pamięci sześć razy więcej miejsca niż zmienna typu `short int`.

```
bool : 1
char : 1
short int : 2
int : 4
long int : 4
float : 4
double : 8
long double : 12
```

```
cout << "bool : " << sizeof(bool) << endl;
cout << "char : " << sizeof(char) << endl;
cout << "short int : " << sizeof(short int) << endl;
cout << "int : " << sizeof(int) << endl;
cout << "long int : " << sizeof(long int) << endl;
cout << "float : " << sizeof(float) << endl;
cout << "double : " << sizeof(double) << endl;
cout << "long double : " << sizeof(long double) << endl;
```

4 Sterowanie programem

Niektóre instrukcje mają szczególne znaczenie. Za ich pomocą możemy sterować wykonywaniem innych poleceń

W rozdziale	strona
Instrukcje warunkowe	**55**
Instrukcja warunkowa if	55
Instrukcja if...else	57
Wielokrotny warunek	59
Instrukcja wyboru switch	60
Warunki zagnieżdżone	62
Operatory logiczne	63
Pustka za instrukcją warunkową	65
Zapętlony program	**66**
Czym jest pętla	67
Pętla typu while	67
Pętla do...while	70
Schemat pętli while i do...while	72
Od 1 do N, czyli pętla for	72
Zagnieżdżone pętle	74
Złożone wyrażenia w pętli for	75
Niebezpieczeństwo złożonych warunków	75
Jak przerwać działanie pętli - instrukcja break	76
Następna iteracja - continue	77
Skocz do, czyli goto	78
Piszemy prawdziwy program	80

Instrukcje znajdujące się w funkcji głównej wykonywane są jedna po drugiej. W ten sposób nie da się jednak tworzyć dużych i praktycznych aplikacji. Dlatego warto poznać polecenia, za pomocą których możemy sterować wykonywaniem kodu źródłowego.

Na początku poznamy instrukcje warunkowe. Są to takie twory języka, które pozwalają komputerowi wykonywać pewien blok instrukcji tylko wtedy, gdy zostaje spełniony określony przez nas warunek.

Oprócz instrukcji warunkowych poznamy również pętle. Dzięki nim każda instrukcja może być wykonana wielokrotnie (oczywiście mamy duży wpływ na liczbę powtórzeń). Szybko się przekonamy, że instrukcje warunkowe i pętle znacznie ułatwiają programowanie oraz oczywiście pozwalają na pisanie lepszych, bardziej rozbudowanych i praktycznych programów.

Ponieważ działanie pętli Komputer ŚWIAT pokazał w formie graficznej, nauczymy się przy okazji rysować proste figury geometryczne.

▼ CD-ROM

plik znajduje się na płycie dołączonej do książki

Nazwa na CD

plik w całości znajduje się na płycie dołączonej do książki, tu pokazany jest tylko jego fragment

Nazwa na CD

Instrukcje warunkowe

W każdym języku programowania istnieją instrukcje pozwalające na sterowanie wykonywaniem programu. Warto je poznać, aby tworzyć naprawdę przydatne aplikacje

Potrafimy już napisać ciąg instrukcji, które będą wykonywane jedna po drugiej. Jednak rzadko która aplikacja działa w taki sposób. Przypomnijmy sobie nasz program obliczający pola trzech figur. Jeśli rzeczywiście za każdym razem będziemy potrzebowali obliczyć wszystkie pola, program spełni nasze oczekiwania. Ale co w wypadku, gdy chcemy obliczyć tylko pole koła? Niestety, będziemy musieli przejść przez dwa pierwsze obliczenia, a to z pewnością nie jest wygodne. Warto więc poznać instrukcje warunkowe, dzięki którym niektóre fragmenty kodu będziemy mogli wykonywać tylko wtedy, gdy zostaną spełnione określone przez nas warunki. W C++ występuje kilka rodzajów instrukcji warunkowych. Na początku poznamy najprostszą z nich – instrukcję if (z ang. jeśli).

Instrukcja warunkowa if

W poprzednim rozdziale poznaliśmy operatory relacji. Jeżeli ich już nie pamiętamy, zajrzyjmy ponownie na stronę 45 i jeszcze raz przeanalizujmy przedstawione tam przykłady. Działanie operatorów relacji polega na wyświetlaniu liczby 0 w wypadku, gdy relacja była fałszywa, oraz 1, gdy była prawdziwa. Zobaczmy teraz, w jaki sposób od wyniku tej relacji uzależnić wykonanie fragmentu kodu.

Napiszemy teraz program, który sprawdza relację dwóch liczb oraz w zależności od tego, która jest większa, wyświetla na ekranie odpowiedni komunikat. Stwórzmy zatem w Dev-C++ nową aplikację konsolową i zapiszmy ją pod nazwą program_18.

Program_18

```
float a, b;
cout << "Podaj liczbe A : ";
cin >> a;
cout << "Podaj liczbe B : ";
cin >> b;
if ( a < b ) cout << "Liczba A jest mniejsza od B." << endl;
if ( a > b ) cout << "Liczba A jest wieksza od B." << endl;
if (a == b) cout << "Liczby A i B sa rowne." << endl;
```

1 Na początku (oczywiście wewnątrz funkcji `main`) wpisujemy polecenia 1 pobierające od użytkownika dwie liczby, które zostają zapisane w zmiennych `a` i `b`.

2 Wpisujemy teraz instrukcję warunkową 2. Jej działanie polega na wyświetleniu na ekranie komunikatu tylko wtedy, gdy wartość zmiennej `a` jest mniejsza od wartości zmiennej `b`.

Ogólna postać instrukcji if

Instrukcję warunkową if możemy ogólnie zapisać w następującej postaci:

if (warunek) instrukcja;

Działanie instrukcji if można rozumieć następująco: Jeśli warunek jest prawdziwy, wykonaj instrukcję. Gdy warunek jest fałszywy, nie rób nic.

► Komputer ŚWIAT radzi

Pamiętajmy, że jeśli w instrukcji if warunkiem jest relacja równości (na przykład 4), wpisujemy operator == (dwa znaki równości). Wpisanie pojedynczego znaku = spowoduje przypisanie do jednej zmiennej wartości drugiej. Operacja taka z pewnością zakończy się sukcesem (zwrócona zostanie wartość 1), a więc warunek zawsze zostanie spełniony, a do tego zmieni się wartość jednej zmiennej. Taki błąd jest trudny do wykrycia i powoduje błędne działanie programu.

```
c:\biblioteczka\program_18\program_18.e
Podaj liczbe A : 12
Podaj liczbe B : 65
Liczba A jest mniejsza od B.
Liczby A i B sa rowne.
```

3 W kodzie umieszczamy jeszcze dwie instrukcje warunkowe 3 i 4. Pierwsza wyświetli komunikat wtedy, gdy to zmienna `b` będzie mniejsza od zmiennej `a`. Druga wyświetli komunikat wtedy, gdy obie liczby będą równe.

4 Aby sprawdzić działanie naszego kodu, kompilujemy i uruchamiamy program. Po wpisaniu dwóch liczb na ekranie pojawi się stosowny komunikat (na przykład `Liczba A jest wieksza od B.` czy `Liczby A i B sa rowne`).

Warunkowe wykonanie kilku poleceń

W przedstawionym przykładzie, przy spełnionym warunku, zostaje wykonana jedna instrukcja `cout`. Jednak nic nie stoi na przeszkodzie, aby warunkowe wykonanie obej-

```
c:\biblioteczka\program_18\program_18.e
Podaj liczbe A : 1.3e3
Podaj liczbe B : 1300
Liczby A i B sa rowne.
```

```
c:\biblioteczka\program_18\program_18.ex
Podaj liczbe A : 12
Podaj liczbe B : 7
Liczba A jest wieksza od B.
```

```
float a, b;
cout << "Podaj liczbe A : ";
cin >> a;
cout << "Podaj liczbe B : ";
cin >> b;

if ( a < b ) {
   cout << "Liczba A jest mniejsza od B." << endl;
   cout << "Roznica pomiedzy zmiennymi wynosi " << b - a << endl;
 }

if ( a >= b ) {
   cout << "Liczba A jest wieksza lub rowna B." << endl;
   cout << "Roznica pomiedzy zmiennymi wynosi " << a - b << endl;
 }
```

Program_19

```
c:\biblioteczka\program_19\program_19.exe
Podaj liczbe A : 20
Podaj liczbe B : 4
Liczba A jest wieksza lub rowna B.
Roznica pomiedzy zmiennymi wynosi 16
```

```
c:\biblioteczka\program_19\program_19.exe
Podaj liczbe A : 8
Podaj liczbe B : 16
Liczba A jest mniejsza od B.
Roznica pomiedzy zmiennymi wynosi 8
```

mowało kilka poleceń naraz. Ogólna postać takiej instrukcji warunkowej wygląda tak:

```
if (warunek) {
 instrukcja1;
 instrukcja2;
...
 instrukcjaN;
}
```

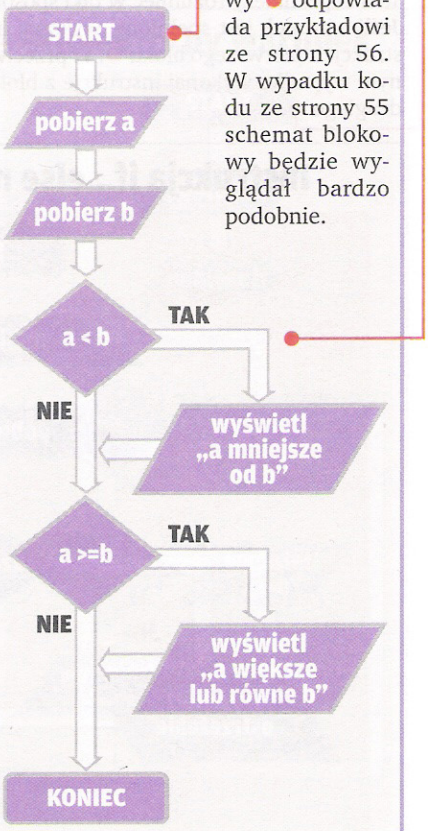
Jeśli więc chcemy, aby w wypadku spełnienia warunku było wykonanych kilka poleceń, wystarczy ująć je w nawiasy klamrowe. Spójrzmy na przykład, którego efektem działania jest widok (gdy `a` jest mniejsze od `b`) lub widok (gdy `a` jest większe lub równe `b`). W przedstawionym przykładzie, za każdym razem sprawdzane są oba warunki, co niepotrzebnie obciąża procesor. Zaraz rozwiążemy ten problem.

Instrukcja if...else

W instrukcji if polecenie (lub polecenia) było wykonywane tylko w wypadku spełnienia określonego warunku. Gdy warunek nie był spełniony, nie było wykonywane żadne polecenie. Dzięki pewnej modyfikacji instrukcji if możemy dodatkowo określić polecenia, które zostaną wykonane, gdy warunek nie zostanie spełniony. Taką zmodyfikowaną instrukcję warunkową określamy jako if...else (else – z ang. w przeciwnym wypadku). Poznajmy tę ważną modyfikację instrukcji warunkowej. Będziemy z niej bardzo często korzystać.

Schemat blokowy instrukcji

W jaki sposób przedstawić instrukcję warunkową na schemacie blokowym? Wystarczy narysować romb, wewnątrz niego wpisać warunek i wyprowadzić z niego dodatkową linię, wskazującą polecenia wykonywane przy spełnionym warunku. Po wykonaniu polecenia tworzymy linię powrotu do dalszej części kodu. Schemat blokowy odpowiada przykładowi ze strony 56. W wypadku kodu ze strony 55 schemat blokowy będzie wyglądał bardzo podobnie.



Ogólna postać instrukcji if...else wygląda tak:

```
if (warunek){
 instrukcja1;
 ...
 instrukcjaN;
} else {
 instrukcjaA;
 ...
 instrukcjaZ;
}
```

Taki zapis należy rozumieć w taki sposób: Jeśli warunek jest spełniony, wykonaj instrukcje z pierwszego bloku. W przeciwnym wypadku wykonaj instrukcje z bloku drugiego.

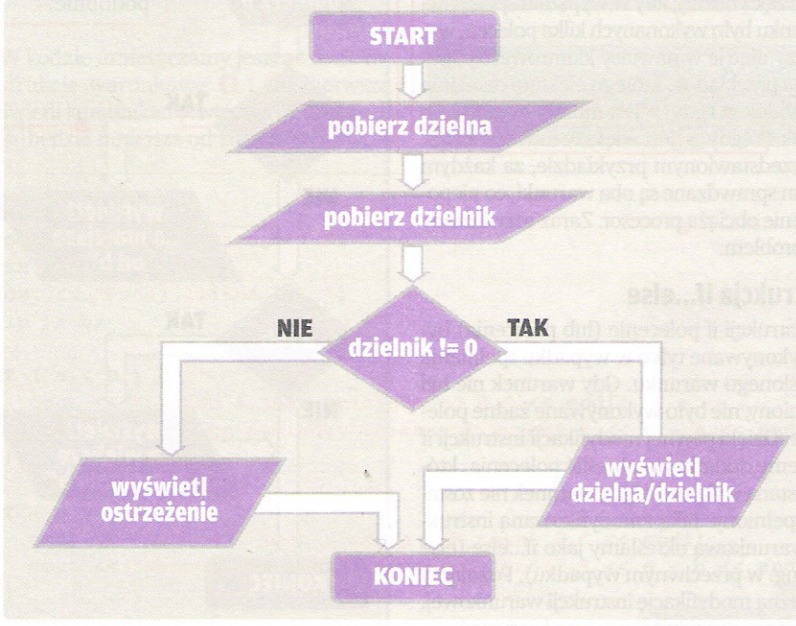
► Komputer ŚWIAT radzi

Jeśli w wypadku spełnienia warunku ma być wykonana tylko jedna instrukcja, nie musimy jej umieszczać w nawiasach klamrowych. Podobnie w wypadku niespełnienia warunku.

Przyjrzyjmy się praktycznemu wykorzystaniu instrukcji warunkowej if...else. Napiszmy program dzielący przez siebie dwie liczby (jego schemat blokowy wygląda tak).

1 Na początku pobieramy od użytkownika dzielną i dzielnik.

Instrukcja if...else na schemacie blokowym



Wiadomo, że dzielenie przez 0 jest niemożliwe i w takim wypadku wystąpi błąd. Za pomocą instrukcji warunkowej sprawdzamy więc, czy dzielnik różny jest od 0. Jeśli tak jest, wyświetlamy wynik dzielenia. W przeciwnym wypadku wyświetlamy komunikat.

Program_20

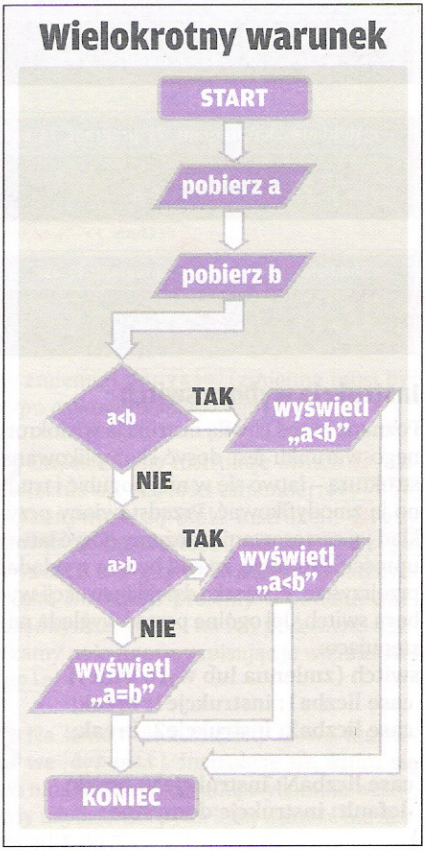
```
float dzielna, dzielnik;
cout << "Podaj dzielna : ";
cin >> dzielna;
cout << "Podaj dzielnik : ";
cin >> dzielnik;

if ( dzielnik != 0) {
     cout << "Iloraz podanych liczb wynosi: ";
     cout << dzielna / dzielnik << endl;
}
else cout << "Nie mozna dzielic przez zero!" << endl;
```

```
c:\biblioteczka\program_20\program_20.ex
Podaj dzielna : 16
Podaj dzielnik : 5
Iloraz podanych liczb wynosi: 3.2
```

```
c:\biblioteczka\program_20\program_20.e
Podaj dzielna : 12
Podaj dzielnik : 0
Nie mozna dzielic przez zero!
```

Wielokrotny warunek

Potrafimy już niektóre instrukcje wykonać tylko wtedy, gdy spełniony jest określony warunek, a inne gdy warunek nie jest prawdziwy. Jednak czasem zachodzi potrzeba wykonania kilku różnych poleceń – każdego przy spełnieniu innego warunku. Taki przykład pokazano przy okazji omawiania instrukcji if na stronie 55. Wtedy zastosowaliśmy po prostu trzy instrukcje warunkowe if. Nie jest to jednak rozwiązanie zbyt eleganckie – jeśli nawet został spełniony warunek z pierwszej instrukcji if i wiadomo, że warunki z kolejnych instrukcji if nie mogą być prawdziwe, i tak są one sprawdzane, co zajmuje niepotrzebnie czas procesora. Dużo lepiej jest w takim wypadku zastosować instrukcję wielokrotnego warunku określaną jako if...else if. Spróbujmy zmodyfikować kod ze strony 56 i umieścić w nim instrukcję if...else if.



Jeżeli nie wiemy, w jaki sposób to zrobić, przyjrzyjmy się przykładowi zastosowania instrukcji wielokrotnego warunku, jego schematowi blokowemu (przedstawiony został na poprzedniej stronie) oraz kilku możliwym rezultatom działania tego programu.

Program_21

```
float a, b;
cout << "Podaj liczbe A : ";
cin >> a;
cout << "Podaj liczbe B : ";
cin >> b;

if ( a < b ) cout << "Liczba A jest mniejsza od B." << endl;
else if ( a > b ) cout << "Liczba A jest wieksza od B." << endl;
else cout << "Liczby A i B sa rowne." << endl;
```

```
c:\biblioteczka\program_21\program_21.exe
Podaj liczbe A : 5
Podaj liczbe B : 10
Liczba A jest mniejsza od B.
```

```
c:\biblioteczka\program_21\program_21.exe
Podaj liczbe A : 3
Podaj liczbe B : 1
Liczba A jest wieksza od B.
```

```
c:\biblioteczka\program_21\program_21.exe
Podaj liczbe A : 234
Podaj liczbe B : 234
Liczby A i B sa rowne.
```

Instrukcja wyboru switch

Poznana przed chwilą instrukcja wielokrotnego warunku jest dosyć skomplikowaną strukturą – łatwo się w niej pogubić i trudno ją zmodyfikować. Przedstawiony przykład jej zastosowania możemy dosyć łatwo uprościć i sprawić, że kod będzie wyglądał przejrzyście. Wszystko dzięki instrukcji wyboru switch. Jej ogólna postać wygląda następująco:

```
switch (zmienna lub wyrażenie){
 case liczba1: instrukcje1; break;
 case liczba2: instrukcje2; break;
...
 case liczbaN: instrukcjeN; break;
 default: instrukcje domyślne;
}
```

Działanie instrukcji wyboru switch można opisać w taki sposób:

- jeśli w nawiasie po słowie kluczowym switch znajduje się wyrażenie, jest ono obliczane,

Komputer ŚWIAT radzi

Wartość wyrażenia lub zmiennej powinna być typu całkowitoliczbowego lub znakowego. Wtedy zamiast liczb po słowie kluczowym case mogą wystąpić symbole znakowe.

- gdy wartość zmiennej lub obliczonego wyrażenia równa jest wartości liczba1, wykonywane są instrukcje1, gdy wartość ta wynosi liczba2, wykonywane są instrukcje2 i tak dalej,

Break

Polecenie break powoduje przerwanie działania instrukcji switch. Powinno się ono znajdować na końcu instrukcji występujących po każdym słowie case. Występowanie po instrukcjach domyślnych (występujących po słowie kluczowym default) nie jest obowiązkowe (instrukcja switch bowiem i tak się po nich kończy). Więcej na temat polecenia break dowiemy się ze strony 76.

```
float x, y, pole;(1)
int decyzja;
cout << "Pole jakiej figury chcesz obliczyć? Podaj numer figury." << endl;
cout << "1 - prostokat" << endl << "2 - trojkat" << endl << "3 - kolo" << endl;(2)
cin >> decyzja;

switch (decyzja){(3)
        case 1:
            cout << "Podaj dlugosci bokow: ";
            cin >> x >> y;
            pole = x * y;
            break;(4)
        case 2:
            cout << "Podaj dlugosc podstawy i wysokosci: ";
            cin >> x >> y;
            pole = 0.5 * x * y;
            break;(5)
        case 3:
            cout << "Podaj promien: ";
            cin >> x;
            pole = 3.14 * x * x;
            break;(6)
        default:
            cout << "Zly wybor. Koniec programu" << endl;;
            pole = 0;(7)
}
cout << "Pole figury wynosi " << pole << endl;(8)
```

Program_22

- gdy nie uda się dopasować wartości zmiennej (lub obliczonego wyrażenia) do żadnej wartości występującej po słowie case, wykonywane są instrukcje domyślne występujące po słowie kluczowym default.

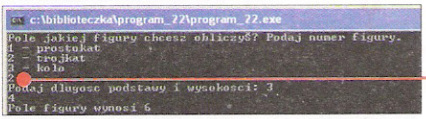
Aby lepiej zrozumieć działanie instrukcji wyboru switch, przyjrzyjmy się przykładowemu kodowi ❶. Jest on zmodyfikowanym programem obliczającym pola różnych figur. Dzięki instrukcji switch użytkownik programu na początku wybiera, jakiej figury pole chce obliczyć.

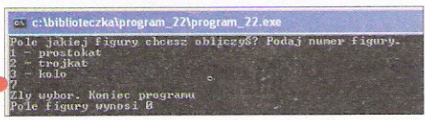
1 Na początku definiujemy trzy zmienne ❶ i wyświetlamy komunikat z prośbą o wybór figury, dla której zostanie obliczone pole ❷. Wybór użytkownika zapisujemy w zmiennej `decyzja` (zmienna musi być typu całkowitoliczbowego).

2 Umieszczamy instrukcję switch ❸. Zależnie od wartości zmiennej `decyzja` wykonywane są bloki instrukcji: ❹ gdy `decyzja` ma wartość 1, ❺ gdy `decyzja` wynosi 2 i ❻ gdy równa się 3. W każdym bloku instrukcji prosimy o podanie niezbędnych danych i na ich podstawie obliczamy pole figury, zapisując je w zmiennej `pole`.

3 Na końcu umieszczamy słowo kluczowe `default`. Instrukcje ❼, które się po nim znajdują, zostaną wykonane wtedy, gdy użytkownik poda inną wartość niż liczba od 1 do 3.

Po umieszczeniu nawiasu klamrowego kończącego instrukcję wyboru switch wyświetlamy obliczone pole figury 8. Uruchamiamy i testujemy nasz program – wpisujemy na przykład poprawny numer figury ● i podajemy dane do obliczeń, a także wpisujemy błędny numer figury ● i sprawdzamy, co się stanie.





► Komputer ŚWIAT radzi

Instrukcja wyboru switch wydaje się doskonałym sposobem na stworzenie programu składającego się z kilku części (tak jak w wymienionym przykładzie). Pamiętajmy jednak, aby nie tworzyć zbyt rozbudowanych instrukcji switch. Kod wtedy stanie się niezbyt przejrzysty i bardzo trudno modyfikowalny. Zamiast skomplikowanych instrukcji switch możemy skorzystać na przykład z funkcji (poznamy je w rozdziale Zapętlony program).

Warunki zagnieżdżone

Czasami zachodzi konieczność wykonania różnych instrukcji w zależności od spełnienia nie jednego, ale kilku warunków. To tak, jakbyśmy kupowali upominek – na początku sprawdzamy, jaką sumę pieniędzy możemy na niego przeznaczyć i w zależności od niej podejmujemy decyzję, jaki to ma być rodzaj prezentu:

<50 zł	50–100	100–200
miś, kwiaty	płyta CD, film DVD	perfumy, kolczyki

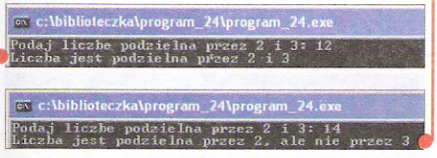
▼ Switch na schemacie

Ponieważ instrukcja switch podobna jest w działaniu do kilku występujących po sobie instrukcji warunkowych if, możemy na schemacie przedstawić ją właśnie w taki sposób. Na przykład schemat blokowy pokazanego przez Komputer ŚWIAT przykładu może wyglądać tak ●. Warto go dokładnie przeanalizować.

W programowaniu występowanie instrukcji warunkowych wewnątrz innych instrukcji warunkowych nazywamy zagnieżdżaniem warunków. Spójrzmy na prosty przykład.

1 Prosimy użytkownika o podanie liczby podzielnej przez 2 i 3 (1). Zadaniem kodu będzie sprawdzenie, czy zmienna `liczba` rzeczywiście jest podzielna przez 2 i 3 i wyświetlenie odpowiedniego komunikatu.

2 Na początku sprawdzamy, czy zmienna `liczba` podzielna jest przez 2 (2). Jeśli tak jest, wykonujemy kolejną instrukcję warunkową, której zadaniem jest sprawdzenie, czy liczba jest podzielna również przez 3 (3). Gdy oba warunki są prawdziwe, wyświetlamy komunikat. Jeśli prawdziwy jest tylko pierwszy warunek (podzielność przez 2), wyświetlamy komunikat.



3 Gdy pierwszy warunek nie jest spełniony, a więc liczba nie jest podzielna przez 2, sprawdzamy, czy liczba podzielna jest przez 3 (4), i jeśli tak jest, wyświetlamy komunikat.

4 Gdy nie uda się dzielenie bez reszty ani przez 2, ani przez 3, wyświetlamy komunikat (5).

Operatory logiczne

W przedstawionym przykładzie do sprawdzenia dwóch warunków (podzielności liczby przez 2 i 3) wykorzystaliśmy dwie instrukcje warunkowe. Rozwiązanie takie, niestety, nie jest ani optymalne, ani czytelne. Na szczęście dzięki istnieniu operatorów logicznych możemy wyrażenia warunkowe łączyć ze sobą i na przykład jednocześnie sprawdzić, czy liczba podzielna jest przez 2 i przez 3.

Jeden z operatorów logicznych już mieliśmy okazję poznać i kilkakrotnie wykorzystać. Był nim operator logicznej negacji oznaczany znakiem wykrzyknika !. Do tej pory jego użycie sprowadzało się do umieszczania go

Program_24

```
int liczba;
cout << "Podaj liczbe podzielna przez 2 i 3: "; (1)
cin >> liczba;
if (liczba % 2 == 0) (2)
{
 (3) if (liczba % 3 == 0) cout << "Liczba jest podzielna przez 2 i 3";
   else cout << "Liczba jest podzielna przez 2, ale nie przez 3";
}
else if (liczba % 3 == 0)(4)
{
   cout << "Liczba jest podzielna przez 3, ale nie przez 2";
}
else
{
    cout << "Liczba nie jest podzielna ani przez 2, ani przez 3"; (5)
}
```

Komputer ŚWIAT radzi

Operator sumy logicznej || można utożsamiać ze słowem LUB. Dla przykładu, wyrażenie A || B jest prawdziwe wtedy i tylko wtedy, gdy A jest prawdziwe lub B jest prawdziwe lub A i B są prawdziwe. Tak więc prawdziwość dowolnego wyrażenia powoduje, że suma logiczna jest prawdziwa.

Operator iloczynu logicznego && można z kolei traktować jako spójnik I. Wyrażenie A && B jest prawdziwe tylko wtedy, gdy prawdziwe jest zarówno A, jak i B.

Spójrzmy na przykład, który jest poprawioną wersją poprzedniego kodu.

1 Po pobraniu liczby od użytkownika w pierwszym warunku (1) sprawdzamy, czy liczba jest podzielna zarówno przez 2, jak i przez 3. Jeśli tak jest, wyświetlamy komunikat.

2 Gdy nie jest spełniony pierwszy warunek, sprawdzamy, czy liczba podzielna jest przez 2 i jednocześnie nie jest podzielna przez 3 (2). Jeśli tak jest, wyświetlamy komunikat.

```
int liczba;
cout << "Podaj liczbe podzielna przez 2 i 3: ";
cin >> liczba;

(1) if ((liczba % 2 == 0) && (liczba % 3 == 0)) cout << "Liczba jest podzielna przez 2 i 3";
(2) else if ((liczba % 2 == 0) && (liczba % 3 != 0)) cout << "Liczba jest podzielna przez 2 ale nie przez 3";
(3) else if ((liczba % 2 != 0) && (liczba % 3 == 0)) cout << "Liczba jest podzielna przez 3 ale nie przez 2";
else cout << "Liczba nie jest podzielna ani przez 2, ani przez 3";
```

Program_25

przed znakiem równości, co w konsekwencji tworzyło operator „różny od", czyli „nie równy". Za pomocą operatora ! możemy również zmieniać wynik relacji. Oto przykład:

(12 == 4 * 3) – prawda

!(12 == 4 * 3) – fałsz

Oprócz operatora negacji warto poznać jeszcze dwa: operator sumy logicznej oznaczany symbolem || i operator iloczynu logicznego oznaczany symbolem &&. Zasadę działania obu Komputer ŚWIAT pokazał w tabeli Operator sumy i iloczynu logicznego. Dodatkowo z przykładowego kodu dowiemy się, jak z operatorów logicznych (na przykładzie &&) korzystać w C++.

```
c:\biblioteczka\program_25\program_25.exe
Podaj liczbe podzielna przez 2 i 3: 18
Liczba jest podzielna przez 2 i 3
```

```
c:\biblioteczka\program_25\program_25.exe
Podaj liczbe podzielna przez 2 i 3: 16
Liczba jest podzielna przez 2 ale nie przez 3
```

3 Warunek (3) sprawdza, czy liczba nie jest podzielna przez 2, ale jest przez 3.

Operator sumy i iloczynu logicznego

wyrażenie A	wyrażenie B	wyrażenie A || B	wyrażenie A && B
prawda	prawda	prawda	prawda
prawda	fałsz	prawda	fałsz
fałsz	prawda	prawda	fałsz
fałsz	fałsz	fałsz	fałsz

Warunek znacznie uproszczony

W C++ istnieje dodatkowy, bardzo ciekawy sposób zapisu instrukcji warunkowej if...else:

warunek? wyrażenie 1: wyrażenie 2

Taki zapis można rozszyfrować następująco: Jeśli spełniony jest warunek, wykonywane jest wyrażenie 1, gdy warunek jest fałszywy, zostaje wykonane wyrażenie 2. Spójrzmy na przykład ●.

1 Po pobraniu wartości zmiennej `liczba` ❶ wykonujemy instrukcję ❷. Jej działanie jest takie: Jeżeli reszta z dzielenia wartości zmiennej `liczba` przez liczbę 2 jest równa 0, wykonywane jest przypisanie do zmiennej `dwa` znaku T, w przeciwnym wypadku do zmiennej `dwa` przypisywany jest znak N.

2 Instrukcja ❸ jest trochę inną formą krótkiego zapisu instrukcji warunkowej. W tym wypadku do zmiennej `trzy` przypisujemy znak T wtedy, gdy reszta z dzielenia liczby przez 3 jest równa 0, lub znak N, gdy reszta jest różna od 0.

```
c:\biblioteczka\program_26\program_26.exe
Podaj liczbe: 12
Czy liczba jest podzielna przez dwa? T
Czy liczba jest podzielna przez trzy? T
```

```
c:\biblioteczka\program_26\program_26.exe
Podaj liczbe: 27
Czy liczba jest podzielna przez dwa? N
Czy liczba jest podzielna przez trzy? T
```

Program_26

```
int liczba;
char dwa, trzy;
cout << "Podaj liczbe: ";
cin >> liczba; ❶

(liczba % 2 == 0) ? dwa = 'T' : dwa = 'N'; ❷
trzy = (liczba % 3 == 0) ? 'T' : 'N'; ❸

cout << "Czy liczba jest podzielna przez dwa? " << dwa << endl;
cout << "Czy liczba jest podzielna przez trzy? " << trzy << endl;
```

3 Po uruchomieniu programu możemy sprawdzić jego działanie, wpisując różne liczby ● (podzielne i niepodzielne przez 2 i 3).

4 Jeśli żaden z wymienionych warunków nie jest spełniony, oznacza to, że zmienna `liczba` nie jest podzielna ani przez 2, ani przez 3.

Pustka za instrukcją warunkową

Jeśli jeszcze nie jesteśmy pewni, kiedy musimy stawiać średniki, a kiedy nie wolno tego robić, zwróćmy uwagę na instrukcję warunkową. Poprawny jest taki jej zapis ●. Jeżelibyśmy niechcący postawili średnik w miejscu ●, instrukcję można by było zapisać w taki sposób ●. Jeśli więc warunek byłby spełniony, wykonywana byłaby instrukcja pusta reprezentowana przez sam średnik. A wtedy polecenie ● wykonywane byłoby zawsze.

```
if (a > 10) cout << "a > 10";
```

```
if (a > 10) ;
cout << "a > 10";
```

```
if (a > 10); cout << "a > 10";
```

Zapętlony program

Chcemy wielokrotnie wykonać fragment kodu? Użyjmy do tego specjalnej konstrukcji – pętli

Poznaliśmy już wiele instrukcji, operatorów i wyrażeń. Możemy już pisać proste, ale przydatne programy. Aby jednak w pełni wykorzystać umiejętność programowania, musimy poznać jeszcze jedną konstrukcję – pętlę. Dzięki niej za pomocą kilku poleceń można wykonywać setki operacji.

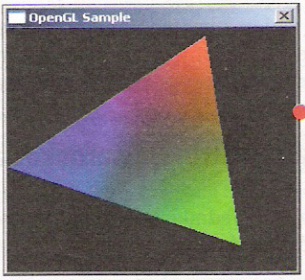
W C++ istnieje kilka rodzajów pętli – poznamy je wszystkie po kolei. Żeby łatwiej zrozumieć pętle, napiszemy też kilka programów przedstawiających w trybie graficznym działanie tej konstrukcji języka. Niestety, w Dev-C++ nie ma prostej metody na tworzenie za pomocą kodu grafiki. Można co prawda z łatwością korzystać z technologii OpenGL, jednak dla początkującego programisty może być ona zbyt skomplikowana i niejasna. Dlatego do wyświetlania grafiki skorzystamy ze specjalnie dostosowanej do programu Dev-C++ biblioteki WinBGIm. Z ramki na stronach 68–69 dowiemy się, w jaki sposób stworzyć projekt programu tak, aby wykorzystywał

OpenGL

OpenGL jest biblioteką graficzną, która służy do tworzenia trójwymiarowej grafiki wektorowej. Jeśli jesteśmy ciekawi, jak wygląda kod programu wykorzystującego OpenGL, możemy otworzyć przykładowy projekt dołączony do Dev-C++. Aby to zrobić, z menu Plik wybieramy pozycję Otwórz projekt lub plik..., w oknie, które się pokazuje, otwieramy katalog c:\dev-cpp\examples\opengl i dwukrotnie klikamy na ikonę OpenGL. Możemy oczywiście skompilować i uruchomić otworzony przez nas projekt – na ekranie zobaczymy okno z obracającym się kolorowym trójkątem. Z pewnością zauważymy, że kod jest dużo bardziej skomplikowany niż omawiane przez Komputer ŚWIAT projekty. Jeśli jednak dobrniemy do końca tej książki i poznamy podstawy programowania, zrozumienie kodu innych programów stanie się dużo łatwiejsze.

Main.cpp

```
// Enable OpenGL

VOID EnableOpenGL( HWND hWnd, HDC * hDC, HGLRC * hRC )
{
  PIXELFORMATDESCRIPTOR pfd;
  int iFormat;

  // get the device context (DC)
  *hDC = GetDC( hWnd );

  // set the pixel format for the DC
  ZeroMemory( &pfd, sizeof( pfd ) );
  pfd.nSize = sizeof( pfd );
  pfd.nVersion = 1;
  pfd.dwFlags = PFD_DRAW_TO_WINDOW |
  PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
  pfd.iPixelType = PFD_TYPE_RGBA;
  pfd.cColorBits = 24;
  pfd.cDepthBits = 16;
  pfd.iLayerType = PFD_MAIN_PLANE;
  iFormat = ChoosePixelFormat( *hDC, &pfd );
  SetPixelFormat( *hDC, iFormat, &pfd );

  // create and enable the render context (RC)
  *hRC = wglCreateContext( *hDC );
  wglMakeCurrent( *hDC, *hRC );
}
```



on możliwości tej biblioteki. Poznamy również kilka ważnych instrukcji umożliwiających po uruchomieniu naszego programu wyświetlenie dodatkowego okna, w którym będziemy mogli rysować punkty, linie czy koła.

Czym jest pętla

Pętla jest blokiem instrukcji, które wykonywane są w kółko (czyli po wykonaniu ostatniej instrukcji z bloku komputer wykonuje ponownie pierwszą instrukcję, później drugą i tak dalej). Oczywiście w większości wypadków takie ciągłe wykonywanie bloku instrukcji spowodowałoby zawieszenie się programu, więc pętlę należy w określonym momencie przerwać. Różnice między typami pętli występującymi w C++ polegają właśnie na sposobie przerywania ich działania. Wielu Czytelników zapewne zada sobie teraz pytanie: Skoro wszystkie pętle działają tak samo, a różnią się jednym szczegółem, to czy warto poznawać wszystkie typy pętli? Odpowiedź jest prosta: warto, ponieważ w niektórych sytuacjach zastosowanie odpowiedniego typu pętli może nam znacznie ułatwić rozwiązanie danego problemu programistycznego.

Pętla typu while

Najprostszym typem pętli jest pętla typu while (z ang. dopóki). Jej ogólną postać można przedstawić tak:

```
while (wyrażenie warunkowe)
{
 instrukcja 1;
 instrukcja 2;
...
 instrukcja N;
}
```

Działanie pętli typu while możemy opisać w taki sposób: Dopóki wyrażenie warunkowe jest prawdziwe, wykonuj instrukcje umieszczone w nawiasach klamrowych. Gdy warunek przestanie być spełniony (będzie fałszem), wykonywanie pętli się kończy i komputer przechodzi do wykonania następnej instrukcji (znajdującej się po pętli while).

► Komputer ŚWIAT radzi

Każde wykonanie bloku instrukcji nazywamy pojedynczym przebiegiem lub iteracją pętli.

Spójrzmy na przykład kodu ●.

```
int x = 0;

while (x < getmaxx())
{
 delay(10);
 putpixel(x, 10, YELLOW);
 cout << x << " ";
 x++;
}
```

Program_28

1 Zgodnie z informacjami z ramki Tworzymy grafikę w Dev-C++ ze stron 68–69 tworzymy nowy projekt (na przykład o nazwie program_28). Zamiast instrukcji rysującej linię umieszczamy instrukcje ●.

2 Na początku deklarujemy zmienną typu całkowitoliczbowego o nazwie `x` i przypisujemy jej wartość 0 ●.

3 Wpisujemy słowo kluczowe `while` i w nawiasach zwykłych podajemy warunek ●. Gdy zmienna `x` będzie mniejsza niż maksymalny rozmiar okna, w którym wyświetlamy grafikę, warunek będzie spełniony (będzie prawdziwy).

4 Dopóki warunek będzie spełniony, będą wykonywane instrukcje ●. Pierwsza wprowadza 10-milisekundowe opóźnienie. Dzięki temu będziemy mogli obserwować kolejne

Tworzymy grafikę w Dev-C++

Abyśmy mogli korzystać z prostych instrukcji umożliwiających rysowanie na ekranie punktów, linii czy okręgów, instalujemy z płyty dołączonej do tej książki biblioteki WinBGIm. Po instalacji na dysku C w folderze biblioteczka\winbgim pojawią się trzy pliki. Musimy je skopiować do odpowiednich katalogów.

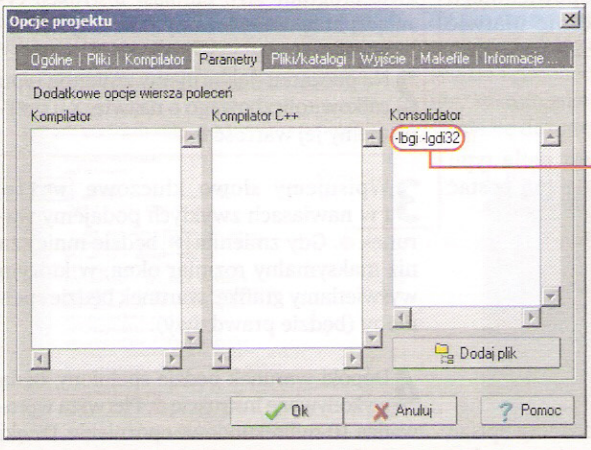
1 Pliki winbgim i winbgim kopiujemy do katalogu c:\dev-cpp\include.

2 Plik libbgi.a kopiujemy do katalogu c:\dev-cpp\lib.

3 Tworzymy nowy projekt (tak jak poprzednio z menu Plik wybieramy Nowy i Projekt..., a następnie wybieramy aplikację konsolową), w pole Nazwa: wpisujemy nazwę naszego projektu (na przykład program_27). Plik projektu zapisujemy w katalogu c:\biblioteczka\program_27.

4 Z menu Projekt... wybieramy pozycję Opcje projektu i w oknie, które się otwiera, klikamy na zakładkę Parametry. W pole Konsolidator wpisujemy ciąg znaków (myślnik, wyraz lbgi, odstęp, myślnik, wyraz lgdi32). Klikamy na przycisk OK. W ten sposób linker dołączy po kompilacji biblioteki graficzne, umożliwiające rysowanie.

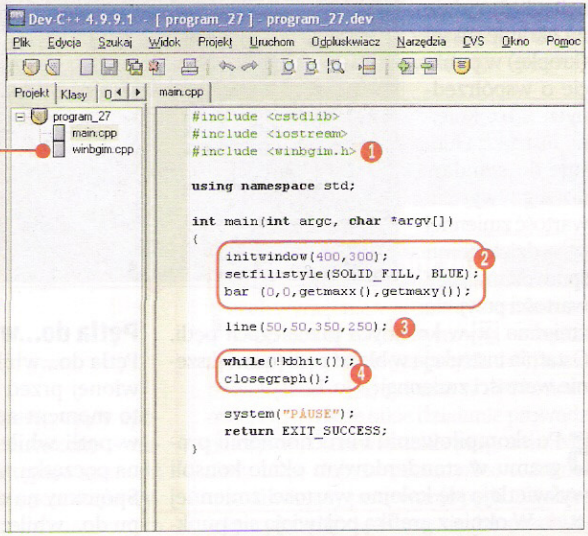
5 Musimy jeszcze dodać do naszego projektu skopiowany przez nas w punkcie 1 plik winbgim. Aby to zrobić, z menu Projekt... wybieramy pozycję Dodaj do projektu. W oknie, które się otworzy, przechodzimy do katalogu c:\dev-cpp\include i dwukrotnie klikamy na ikonę winbgim (zwróćmy uwagę, aby wybrać plik o tej nazwie z ikoną). W oknie projektu pojawi się nowy plik – jeśli chcemy, możemy na niego dwukrotnie kliknąć i podejrzeć jego zawartość.



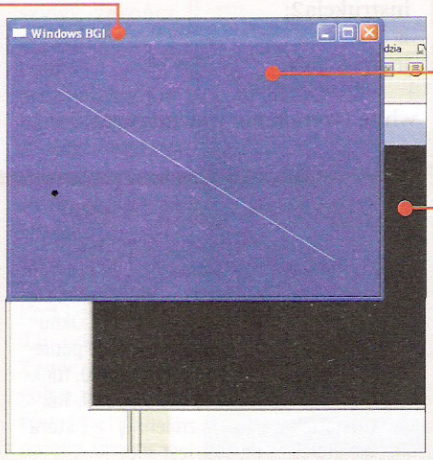
6 Pozostało nam już tylko do projektu wstawić skopiowany wcześniej plik winbgim. Zrobimy to za pomocą dyrektywy `include`, którą wstawiamy do kodu naszego programu ❶.

7 W funkcji `main` możemy już umieszczać polecenia umożliwiające rysowanie punktów czy linii. Wcześniej jednak wpisujemy instrukcje ❷ oraz ❹ (między tymi dwoma blokami umieścimy kod nasze-

go programu). Zadaniem instrukcji z bloku 2 jest wyświetlenie okna o rozdzielczości 400 na 300 oraz wypełnienie całego okna kolorem niebieskim. Polecenia z bloku 4 wstrzymują zamknięcie okna z grafiką do chwili naciśnięcia przez nas dowolnego przycisku oraz zamykają okno.



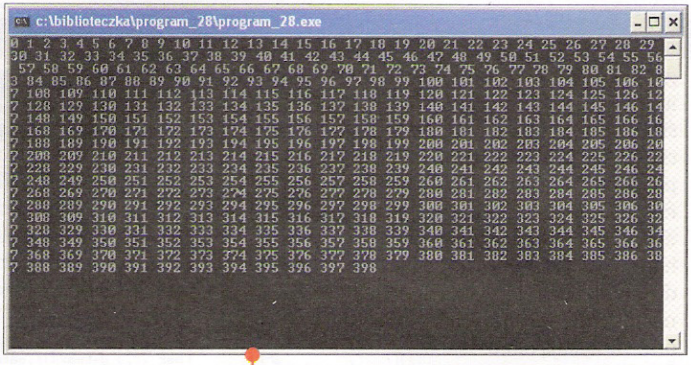
8 Aby sprawdzić działanie naszego programu, wpiszmy instrukcję rysującą linię 3. Podane parametry oznaczają, że zostanie narysowana linia od punktu 50,50 do punktu 350,250. Po skompilowaniu i uruchomieniu naszego programu na ekranie widzimy . Wciskając dowolny klawisz, zamykamy okno , a wciśnięcie kolejnego klawisza powoduje zamknięcie okna .



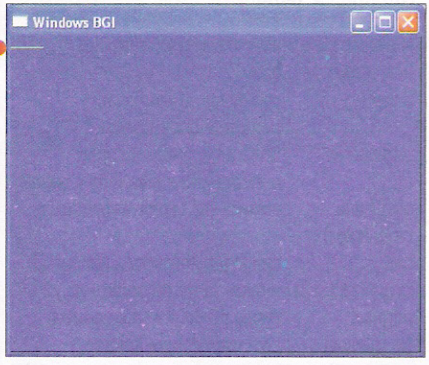
Niektóre polecenia z biblioteki WinBGIm

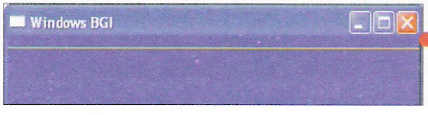
polecenie	opis
delay (x)	opóźnienie wykonywania programu o x milisekund
initwindow (x, y)	uruchamia graficzne okno o rozmiarach x pikseli na y pikseli
setfillstyle (styl, kolor)	ustawia styl i kolor wypełnienia
bar (x1, y1, x2, y2)	rysuje prostokąt rozciągnięty od punktu x1,y1 do punktu x2,y2
putpixel (x, y, kolor)	rysuje punkt o współrzędnych x,y i ustawionym kolorze
getmaxx ()	pobiera rozmiar okna w poziomie
getmaxy ()	pobiera rozmiar okna w pionie
circle (x, y, r)	w punkcie x,y rysuje okrąg o promieniu r

przebiegi pętli „na żywo". Druga instrukcja rysuje na ekranie piksel (kropkę) w punkcie o współrzędnych x,10. Kolejna instrukcja kieruje do standardowego wyjścia wartość zmiennej `x` – dzięki temu sprawdzimy, jakie wartości przyjmie zmienna `x` w kolejnych przebiegach pętli. Ostatnia instrukcja w bloku pętli to zwiększenie wartości zmiennej `x` o 1.



5 Po skompilowaniu i uruchomieniu programu w standardowym oknie konsoli wyświetlają się kolejno wartości zmiennej `x`. W oknie z grafiką pojawiają się punkty – efekt będzie przypominał rysowanie linii.





Pętla do...while

Pętla do...while jest modyfikacją przedstawionej przed chwilą pętli while. Różnica to moment sprawdzania warunku pętli – w pętli while warunek jest sprawdzany na początku, w pętli do...while – na końcu. Spójrzmy na ogólny kod działania pętli typu do...while:

```
do
{
 instrukcja1;
 instrukcja2;
...
 instrukcjaN;
}
while (wyrażenie warunkowe);
```

► Komputer ŚWIAT radzi

Jeśli przyjrzymy się wyświetlanym wartościom zmiennej `x`, zauważymy, że program wygenerował liczby od 0 do 398. Zgadza się to z warunkiem `x < getmaxx()`. Okno grafiki ma 400 pikseli szerokości, ale ponieważ kolejne piksele numerujemy od 0, funkcja `getmaxx()` zwraca wartość 399. Tak więc ostatnią wartością zmiennej `x`, która będzie mniejsza od 399, jest 398.

▶ Komputer ŚWIAT radzi

Zwróćmy uwagę, że w pętli do...while warunek sprawdzany jest już po wykonaniu instrukcji. W związku z tym instrukcje niezależnie od prawdziwości warunku będą wykonane co najmniej jeden raz. Na przykład uruchomienie kodu spowoduje wyświetlenie na ekranie jednego napisu Komputer SWIAT mimo ewidentnie fałszywego warunku.

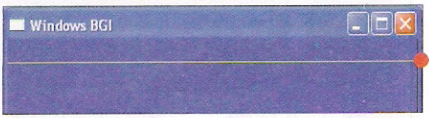
```
do
{
    cout << "Komputer SWIAT";
}
while (0);
```

Pętla wykonywana jest w następujący sposób: Wykonaj instrukcje umieszczone w nawiasach klamrowych, a następnie sprawdź warunek. Jeśli jest prawdziwy, ponownie wykonaj instrukcje i znowu sprawdź warunek. Gdy warunek jest fałszywy, zakończ wykonywanie pętli.

Sprawdźmy, jak będzie wyglądał i jak będzie działał poprzedni przykład, jeśli zamiast pętli while użyjemy pętli do...while. Kod będzie bardzo podobny. Zwróćmy szczególną uwagę na średnik, który należy umieścić na końcu linii ze słowem kluczowym `while` (przy pętli typu while średnika nie można postawić). Po skompilowaniu i uruchomieniu kodu na ekranie zobaczymy identyczny jak w poprzednim przykładzie wynik.

```
int x = 0;
do
{
 cout << x << " ";
 putpixel(x,20,YELLOW);
 delay(10);
 x++;
}
while (x < getmaxx());
```

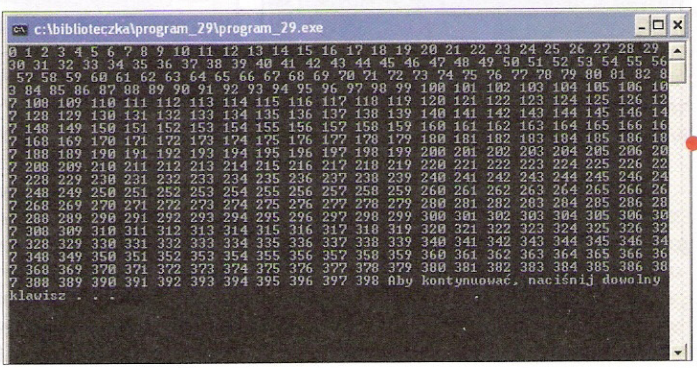
Program_29

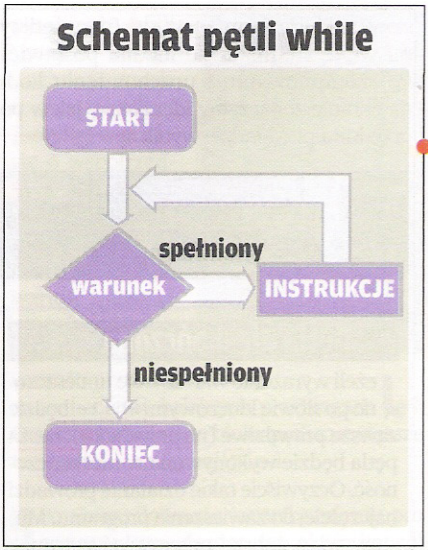


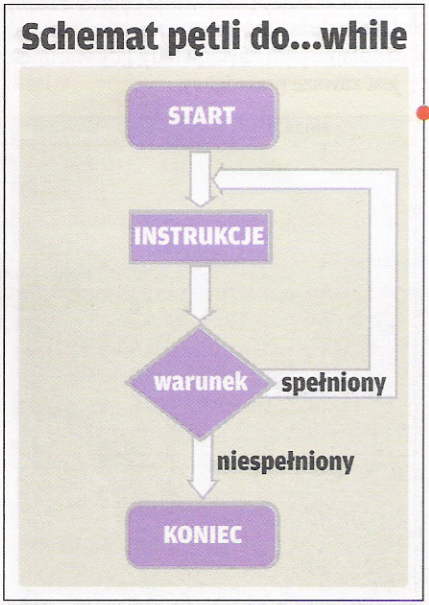
▼ Pętla nieskończona

Jeżeli wyrażenie warunkowe umieszczone po słowie kluczowym `while` będzie zawsze prawdziwe (na przykład), nasza pętla będzie wykonywana w nieskończoność. Oczywiście takie działanie prowadzi najczęściej do zawieszenia programu. Musimy więc dobrze przemyśleć warunek umożliwiający przerwanie pętli. Z dalszej lektury książki dowiemy się, jak przerwać działanie pętli nawet wtedy, gdy warunek jest zawsze prawdziwy.

```
while (1)
{
    cout << "Komputer SWIAT";
}
```







Schemat pętli while i do...while

Zrozumienie działania pętli typu while i do...while ułatwi nam z pewnością schemat blokowy, który możemy rozrysować. W wypadku pętli while wygląda on tak. Jak widzimy, jest dosyć prosty. Zwróćmy uwagę, że najpierw sprawdzane jest wyrażenie warunkowe i dopiero wtedy, gdy jest ono prawdziwe, wykonywane są instrukcje umieszczone wewnątrz pętli. Schemat blokowy pętli do...while wygląda tak. Tak więc widać, że najpierw wykonywane są instrukcje, a dopiero później sprawdzany jest warunek zakończenia pętli.

Od 1 do N, czyli pętla for

Oprócz wspomnianych pętli while i do...while w języku C++ (a także chyba w każdym innym języku programowania) występuje pętla typu for. Ogólny schemat pętli typu for można przedstawić następująco:

```
for (wyrażenie inicjujące; wyrażenie warunkowe; wyrażenie zmieniające)
{
 instrukcja1;
 instrukcja2;
...
 instrukcjaN;
}
```

Wykonanie instrukcji for rozpoczyna się od wyrażenia inicjującego, w którym inicjowana jest pewna zmienna (najczęściej nazywana `i` lub `j`). Wyrażenie inicjujące wykonywane jest tylko jeden raz – na samym początku działania pętli for.

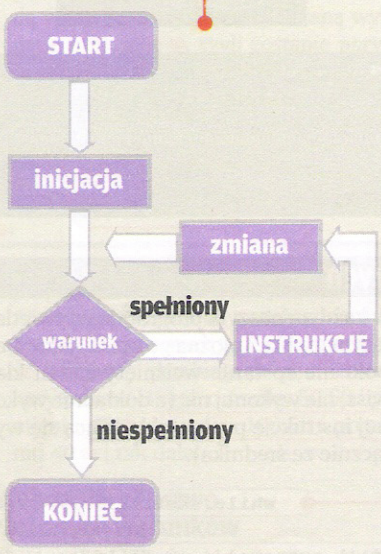
Zaraz po wykonaniu wyrażenia inicjującego oraz po udanym zakończeniu każdego przebiegu pętli sprawdzane jest wyrażenie warunkowe (zazwyczaj zawiera ono w sobie zainicjowaną przed chwilą zmienną). Jeśli wyrażenie warunkowe zwraca wartość różną od zera (czyli jest prawdzi-

we), zostają wykonane instrukcje umieszczone w nawiasach klamrowych. Jeżeli wyrażenie warunkowe zwraca wartość 0 (jest fałszywe), wykonywanie pętli zostaje przerwane.

Wyrażenie zmieniające wykonywane jest po każdym zakończeniu wykonywania instrukcji z nawiasów klamrowych, ale przed sprawdzeniem wyrażenia warunkowego. Zadaniem wyrażenia zmieniającego jest zmiana wartości zmiennej zainicjowanej w wyrażeniu inicjującym.

Schemat instrukcji for

Pętla typu for wydaje się dosyć skomplikowaną konstrukcją języka C++. Jeśli nie do końca rozumiemy jej działanie, przyjrzyjmy się schematowi blokowemu instrukcji for.



Zobaczmy działanie pętli for w praktyce. Zmieńmy kod poprzedniego przykładu w taki sposób, aby wykorzystywać w nim pętlę for.

```
for (int i = 0; i < getmaxx(); i++)
{
   delay(10);
   putpixel(i,10,YELLOW);
   cout << i << " ";
}
```

Program_30

1 W nawiasach po słowie kluczowym `for` wpisujemy trzy wyrażenia: inicjujące, warunkowe i zmieniające.

2 Zadaniem pierwszego jest inicjacja zmiennej `i` wartością 0.

3 Drugie wyrażenie określa warunek przejścia do kolejnego przebiegu pętli – w naszym wypadku kolejna iteracja pętli zostaje wykonana, kiedy `i` jest mniejsze od maksymalnego rozmiaru okna (czyli gdy zapis `i < xgetmaxx()` jest prawdziwy).

4 Wyrażenie zmieniające powoduje inkrementację (zwiększenie o 1) wartości zmiennej `i`.

5 Dopóki wyrażenie warunkowe jest prawdziwe, komputer wykonuje instrukcje, czyli następuje opóźnienie (abyśmy mogli obserwować działanie pętli), zostaje postawiony piksel w punkcie określonym przez zmienną `i` i liczbę 10 oraz na standardowym wyjściu jest wyświetlana wartość zmiennej `i`.

6 Po skompilowaniu i uruchomieniu programu na ekranie widać okno z powstającą z punktów linią oraz okno z kolejnymi wartościami zmiennej `i`. Oba okna są takie same, jak po wykonaniu programu wykorzystującego pętlę while oraz do...while.

Zagnieżdżone pętle

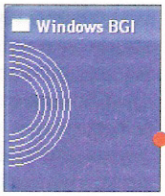
Pętle, podobnie jak instrukcje warunkowe, można zagnieżdżać, czyli w bloku instrukcji jednej pętli umieszczać inną pętlę. Stwórzmy więc program, który zademonstruje taką złożoną konstrukcję.

```
for (int x = 0; x < getmaxx(); x += 50)1
{
    for (int r = 50; r > 0; r -= 5)2
    {
        delay(50);
        circle(x,50,r);
    }
}
```

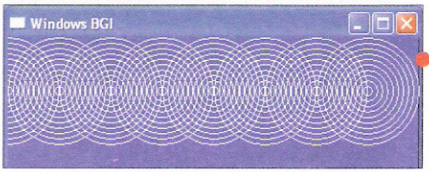
Program_31

1 Pierwsza pętla 1 jest wykonywana tak długo, jak długo zmienna `x` jest mniejsza od maksymalnego rozmiaru okna (do zmiennej `x` w każdym przebiegu pętli jest dodawana liczba 50 – tak więc zmienna `x` kolejno przyjmuje wartości: 0, 50, 100 i tak dalej).

2 W każdym przebiegu pętli 1 jest uruchamiana pętla 2. Jej zadaniem jest wyświetlanie okręgu, którego środek jest oddalony od lewej krawędzi okna o wartość zmiennej `x`. Promień okręgu (zapisany w zmiennej `r`) w pierwszym przebiegu pętli jest równy 50, w drugim 45, potem 40, 35 i tak dalej.



3 Po zakończeniu wykonywania pętli 2 (czyli po narysowaniu 10 okręgów o różnych promieniach) pętla 1 rozpoczyna swój drugi przebieg, w którym znowu jest rysowanych 10 okręgów. Po skompilowaniu i uruchomieniu programu widzimy (obrazki przedstawiają efekty działania programu w różnych momentach jego wykonywania).



Pętle bez nawiasów klamrowych

Jeśli w pętli typu while lub for wykonywana jest tylko jedna instrukcja, możemy pominąć nawiasy klamrowe. Pamiętajmy jednak, aby po instrukcji wstawić średnik. W wypadku for pętla z jedną instrukcją mogłaby wyglądać tak, a przy pętli typu while tak. Jak pewnie zauważyliśmy, zapis ten znajduje się w każdym naszym programie wyświetlającym grafikę i można go opisać jako: Dopóki nie zostanie wciśnięty żaden klawisz, nie wykonuj nic (a dokładnie wykonuj instrukcję pustą – składającą się wyłącznie ze średnika).

```
while(!kbhit());
```

```
for (int i = 0; i < getmaxx(); i++) putpixel(i,10,YELLOW);
```

Złożone wyrażenia w pętli for

W wypadku zagnieżdżenia dwóch pętli druga pętla wykonywana jest w całości w każdym przebiegu pętli pierwszej. Jednak niekiedy zachodzi potrzeba równoczesnego wykonywania dwóch pętli. Taki efekt można uzyskać, stosując złożone wyrażenia w nagłówku pętli for. Spójrzmy na przykład ●, w którym w pierwszej pętli do każdego wyrażenia (inicjującego, warunkowego, zmieniającego) po przecinku dodano odpowiednie wyrażenie z pętli drugiej. Program będzie działał w taki sposób:

- zostaną zainicjowane zmienne `x` i `r` – pierwszej zostanie przypisana wartość 0, drugiej – 50,
- ponieważ oba wyrażenia warunkowe będą w tej chwili spełnione, zostaną wykonane instrukcje ●, czyli zostanie narysowany okrąg o promieniu 50, którego środek znajduje się w punkcie 0,50 ●.

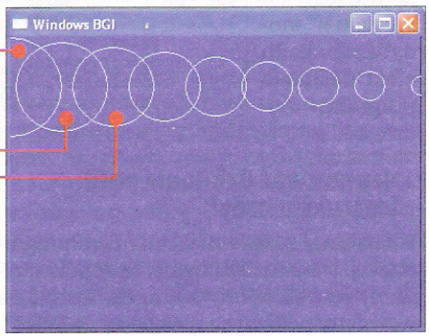
```
for (int r = 50, kolor = 0; r > 0, kolor <= 15; r -= 5, kolor++)
{
        delay(50); 1
        setcolor(kolor); 2
        circle(200,200,r); 3
        cout << r << "-" << kolor << endl; 4
}
```

Program_33

```
for (int x = 0, r = 50; x < getmaxx(), r > 0; x += 50, r -= 5)
{
        delay(50);
        circle(x,50,r);
}
```

Program_32

- po wykonaniu instrukcji ● wartość zmiennej `x` zostanie zwiększona o 50, a wartość zmiennej `r` zmniejszona o 5.
- w kolejnych przebiegach pętli zostaną narysowane okręgi: o środku 50,50 i promieniu 45 ●, o środku 100,50 i promieniu 40 ● i tak dalej.



Niebezpieczeństwo złożonych warunków

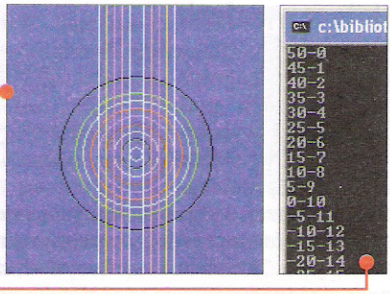
Złożone wyrażenia warunkowe w definicji pętli for stanowią pewne zagrożenie. Spójrzmy na przykład kodu ●, w którym występuje takie niebezpieczeństwo.

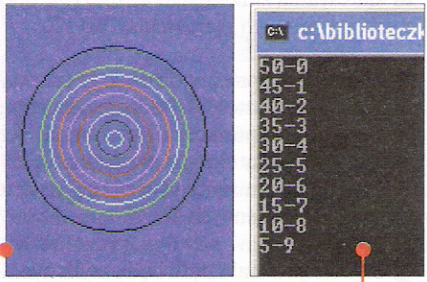
1 Wykonanie pętli rozpoczyna się od zadeklarowania zmiennych `r` i `kolor` oraz zainicjowania ich wartościami 50 i 0.

2 W ciele pętli oprócz opóźnienia 1 ustawiamy kolor o wartości zapisanej w zmiennej `kolor` 2 i w punkcie o współrzędnych 200,200 rysujemy okrąg o promieniu `r` 3. Dla naszej informacji wyświetlamy jeszcze wartość obu zmiennych 4.

3 Instrukcje z ciała pętli będą wykonywane, dopóki spełniane będą warunki ● (czyli gdy promień będzie większy od 0, a `kolor` mniejszy lub równy 15, przy czym promień w każdym przebiegu pętli zmniejszamy o 5, a zmienną `kolor` zwiększamy o 1).

Aby sprawdzić, jak zadziała taka pętla, kompilujemy i uruchamiamy program. Na ekranie widzimy ●. Jak wytłumaczyć taki efekt? Wystarczy zobaczyć wartości zmiennych w kolejnych przebiegach ●. Jak widzimy, promień w pewnym momencie przyjmował wartości ujemne mimo warunku przejścia do następnej iteracji r > 0. Dlaczego tak się stało? Ponieważ pętla była wykonywana aż do momentu, w którym wszystkie wyrażenia warunkowe pętli stały się fałszywe.





Jednak w ramce Pętla nieskończona na stronie 71 Komputer ŚWIAT napisał, że jest możliwe przerwanie działania pętli niezależnie od wyrażenia warunkowego. Do tego celu służy instrukcja break. Wielu Czytelników z pewnością pamięta, że korzystaliśmy już z niej przy instrukcji wyboru switch, gdzie również służyła do przerywania działania instrukcji, w której ją umieszczono. Spójrzmy zatem na przykład ●.

Program_35

```
int r = 5;
while (1)
{
        if (r>100) break; 1
        delay(10);
        circle(200,150,r); 2
        r+=5; 3
}
```

Jeśli więc chcemy, aby pętla została zakończona wtedy, gdy którekolwiek z wyrażeń warunkowych zacznie zwracać wartość 0 (fałsz), zastosujmy operator iloczynu logicznego ● (jak pamiętamy, zwraca on wartość 1 tylko i wyłącznie wtedy, gdy oba wyrażenia zwracają wartość 1). Po takiej modyfikacji kodu efekt działania programu będzie taki ●.

```
for (int r = 50, kolor = 0; (r > 0) && (kolor <= 15); r -= 5, kolor++)
{
        delay(50);
        setcolor(kolor);
        circle(200,200,r);
        cout << r << "-" << kolor << endl;
}
```

Program_34

Jak przerwać działanie pętli – instrukcja break

Poznaliśmy zasadę działania pętli programowej. Dowiedzieliśmy się, że dopóki spełniony jest określony warunek, instrukcje wewnątrz pętli wykonywane są „w kółko".

1 Definiujemy zmienną x (przypisujemy jej wartość 5) i tworzymy pętlę nieskończoną while.

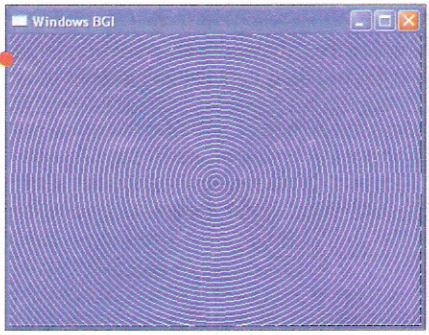
2 Instrukcja 2 rysuje w punkcie o współrzędnych 200,150 okrąg o promieniu

► Komputer ŚWIAT radzi

Oczywiście instrukcja `break` przerywa działanie każdej pętli, nie tylko nieskończonej. Tak więc wewnątrz każdej pętli możemy umieścić dodatkowy warunek zakończenia jej działania.

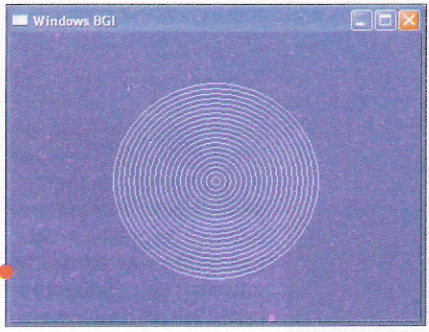
równym wartości zmiennej `r`, która w każdym przebiegu jest zwiększana o 5 ❸.

3 Instrukcja warunkowa ❶ sprawdza, czy promień jest już większy od 100. Jeśli tak jest, wykonywana jest instrukcja `break` powodująca przerwanie działania pętli. Gdybyśmy nie wpisali tej instrukcji, na ekranie zobaczylibyśmy widok ●, a program by się zawiesił. Dzięki instrukcji `break` możemy w dowolnym momencie przerwać działanie pętli ●.



► Komputer ŚWIAT radzi

Instrukcja `break` przerywa działanie tylko tej pętli, w ciele której została wywołana. Jeśli więc wywołamy `break` w pętli zagnieżdżonej w innej pętli, zostanie przerwane działanie tylko tej wewnętrznej pętli. O tym, jak skutecznie przerwać działanie nawet zagnieżdżonych pętli, przeczytamy na następnej stronie.



Następna iteracja - continue

Poznana przed chwilą instrukcja `break` przerywa całkiem działanie pętli. Jednak w niektórych wypadkach potrzebne jest przerwanie tylko aktualnego przebiegu pętli, czyli przejście do następnej iteracji pętli. Do tego celu służy instrukcja continue. Jej działanie poznamy na przykładzie ●.

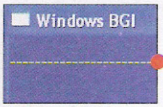
```
for (int x = 0; x < getmaxx(); x++)
{
    if (x%5==0) continue;
    putpixel (x,20,YELLOW);
    cout << x << endl;
}
```

Program_36

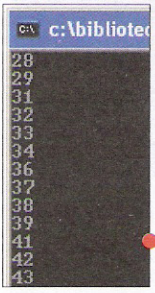
1 Tworzymy pętlę for, w której zmienna `x` w kolejnych przebiegach zmienia się o wartość 1 od 0 do maksymalnego rozmiaru okna.

2 W ciele funkcji umieszczamy instrukcję warunkową ●. Kiedy wartość zmiennej `x` jest podzielna przez 5 (dzielenie `x` przez 5 będzie bez reszty), zostaje wykonana instrukcja `continue`. Powoduje ona przerwanie aktualnego przebiegu pętli i rozpoczęcie wykonywania kolejnego (zmienna `x` jest zwiększana, zostaje sprawdzony warunek zakończenia pętli i instrukcje z ciała funkcji zostają ponownie wykonane).

3 Gdy przebieg pętli nie zostaje przerwany, jest rysowany piksel w punkcie o współrzędnych x,20.



Efektem działania tego programu jest linia przerywana (przerwa następuje w co piątym punkcie linii). Aby przekonać się, że przebieg pętli zostaje przerywany wtedy, gdy x jest podzielne przez 5, spójrzmy na wygenerowany dzięki instrukcji cout zapis – brakuje w nim liczb podzielnych przez 5.



Skocz do, czyli goto

W C++ występuje pewna instrukcja, której zdecydowanie nie zaleca się wykorzystywać. Jest to instrukcja goto pozwalająca na skok z jednego do innego miejsca programu. Dlaczego nie powinno się korzystać z instrukcji goto? Ponieważ są dużo lepsze sposoby rozwiązania problemów programistycznych niż goto – na przykład funkcje (więcej o funkcjach dowiemy się z następnego rozdziału). No, ale skoro nie należy używać instrukcji goto, to po co o niej w ogóle pisać? Z dwóch powodów. Po pierwsze dlatego, że warto wiedzieć, jak działa instrukcja goto – zawsze możemy się na nią natknąć w cudzych programach. Po drugie dlatego, że może się zdarzyć, że zastosowanie goto okaże się lepszym rozwiązaniem niż każde inne. Poznamy teraz taką sytuację i na jej przykładzie zobaczymy, jak działa instrukcja goto.

1 Program oblicza pole i obwód koła. Możemy wielokrotnie podawać różne promienie i wybierać, co ma zostać obliczone bez ponownego uruchamiania programu.

2 W nieskończonej pętli 1 pobrana zostaje od użytkownika długość promienia koła 2 i uruchomiona druga nieskończona

```
char znak1;
float r;
while(1) 1
{
    cout << "Podaj promien kola: ";
    cin >> r; 2
    while(1) 3
    {
        cout << "1 - oblicz pole, 2 - oblicz obwod, " << endl;
        cout << "3 - podaj inny promien, q - zakoncz program" << endl;
        cin >> znak1;
        if (znak1=='1') cout << "Pole wynosi " << 3.14 * r * r << endl;
        else if (znak1=='2') cout << "Obwod wynosi " << 2 * 3.14 * r << endl;
        else if (znak1=='3') break; 4
        else if (znak1=='q') goto koniec; 5
    }
}

koniec:
cout << "Koniec programu!" << endl << endl; 6
```

Program_37

pętla 3, w której użytkownik decyduje, co chce obliczyć.

Wybranie przez niego cyfry 1 powoduje obliczenie pola koła, wybranie cyfry 2 – jego obwodu. Gdy użytkownik wybiera cyfrę 3, zostaje wykonana instrukcja `break` 4 powodująca wyjście z aktualnej pętli 3 i wyświetlenie ponownej prośby o podanie promienia. Ale jak z wnętrza pętli 3 opuścić jednocześnie obie pętle? Wystarczy skorzystać z instrukcji `goto` 5. Wtedy po wciśnięciu klawisza Q wykonanie programu zostaje przeniesione do pierwszej instrukcji znajdującej się po etykiecie o nazwie `koniec:`, czyli do instrukcji 6, która powoduje wyświetlenie na ekranie komunikatu.

```
c:\ c:\biblioteczka\program_37\program_37.exe
Podaj promien kola: 10
1 - oblicz pole, 2 - oblicz obwod,
3 - podaj inny promien, q - zakoncz program
1
Pole wynosi 314
1 - oblicz pole, 2 - oblicz obwod,
3 - podaj inny promien, q - zakoncz program
2
Podaj promien kola: 2
1 - oblicz pole, 2 - oblicz obwod,
3 - podaj inny promien, q - zakoncz program
q
Koniec programu!
```

Pętle while – warunek i zmiana w jednym

W pętlach typu for wyrażenia inicjujące, warunkowe oraz zmieniające znajdują się w nagłówku pętli. Dzięki temu kod jest bardzo przejrzysty. W naszych przykładach pętli while i do... while w nagłówku znajduje się jedynie wyrażenie warunkowe, a zmienna iteracyjna zmieniana jest w ciele pętli. Jednak nie ma przeszkód, aby przy sprawdzaniu warunku przejścia do następnej iteracji pętli zmieniać jednocześnie wartość zmiennej iteracyjnej. Jak to robić? W wypadku najczęściej występującej zmiany – inkrementacji – możemy postąpić na przykład tak (dla pętli while) lub tak (w wypadku pętli do...while). Taka pętla while działa następująco: na początku sprawdzany jest warunek i < 10 i niezależnie od wyniku tego sprawdzenia zmienna i zwiększana jest o 1. Jeżeli warunek jest prawdziwy, wykonywane są instrukcje z ciała pętli. Należy zwrócić uwagę, że warunek przed pierwszą iteracją sprawdzany jest przy wartości i wynoszącej 0, a instrukcje z ciała pętli są wykonywane przy i równym 1. Tak więc efektem działania takiej pętli jest wynik (inny niż w wypadku pętli ze zmianą i w ciele pętli). W wypadku pętli do...while zmienna i przyjmuje wartości od 0 do 9. Dlaczego? Spróbujmy to przeanalizować we własnym zakresie.

```
int i = 0;
while(i++ < 10){
   cout << i << " ";
}
```

```
int i = 0;
do {
   cout << i << " ";
}
while(i++ < 9);
```

```
int i = 0;
while(i < 10){
   cout << i << " ";
   i++;
}
```

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9
```

Piszemy prawdziwy program

Dzięki wykorzystaniu instrukcji warunkowych, pętli programowych oraz instrukcji sterujących break i continue możemy z naszego prostego programu obliczającego pola różnych figur stworzyć funkcjonalną aplikację.

1 Na początku deklarujemy kilka niezbędnych zmiennych (1) typu znakowego i zmiennoprzecinkowego.

2 Następnie tworzymy nieskończoną pętlę while (2) – dzięki temu program będzie działał „w kółko" (później stworzymy oczywiście możliwość jego opuszczenia).

3 Wyświetlamy menu naszego programu (3) i w zmiennej `znak1` zapisujemy wybór użytkownika.

4 Korzystając z instrukcji wyboru switch, w zależności od wybranej figury pobie-

```
int main(int argc, char *argv[])
{
    char znak1, znak2;                                          (1)
    float a, b, pole;
    while(1)                                                    (2)
    {
        cout << "Wybierz figure: " << endl;
        cout << "1 - prostokat, 2 - kolo, 3 - trojkat, 0 - koniec " << endl;   (3)
        cin >> znak1;

        switch (znak1){
            case '1': cout << "Podaj dlugosci bokow: ";
                      cin >> a >> b; pole = a * b; break;       (4)
            case '2': cout << "Podaj promien: ";
                      cin >> a; pole = 3.14 * a * a; break;
            case '3': cout << "Podaj dlugosc podstawy i wysokosc: ";
                      cin >> a >> b; pole = 0.5 * a * b; break;
            case '0': goto koniec;                              (5)
            default: continue;                                  (6)
        }
        cout << "Pole figury wynosi " << pole << endl;          (7)
        cout << "Wykreslic figure? (t/n): ";                    (8)
        cin >> znak2;
        if (znak2=='t'){
            initwindow (400,400);                               (9)
            switch (znak1){
                case '1': bar(int(200-(a/2)),int(200-(b/2)),int(200+(a/2)),int(200+(b/2))); break;
                case '2': circle(200,200,(int)a); break;
                case '3': line(int(200-(a/2)),int(200-(b/2)),int(200+(a/2)),int(200-(b/2)));
                          line(int(200-(a/2)),int(200-(b/2)),int(200-(a/2)),int(200+(b/2)));
                          line(int(200-(a/2)),int(200+(b/2)),int(200+(a/2)),int(200-(b/2)));
                          break;
            }
            while(!getch());
            closegraph();
        }
    }
    koniec:
    cout << "Koniec programu!" << endl << endl;
```

Program_38

ramy dane dotyczące figury oraz obliczamy pole wybranej figury ❹.

5 Gdy użytkownik zdecydował o zakończeniu programu (wybrał 0) ❺, przenosimy działanie programu do etykiety `koniec:`. Wtedy wyświetlony zostaje komunikat i działanie programu się kończy. Gdy użytkownik wybiera inny klawisz ❻, za pomocą instrukcji `continue` przechodzimy do kolejnego przebiegu funkcji (czyli jeszcze raz wyświetlamy menu naszego programu).

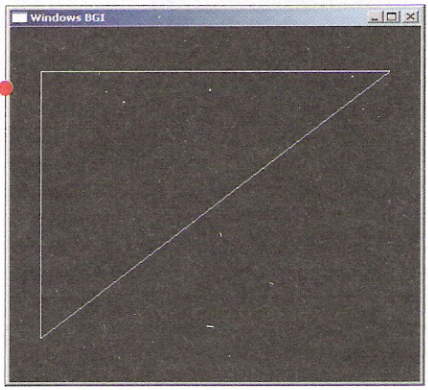
```
C:\Biblioteczka\program_38\program_38.exe
Wybierz figure:
1 - prostokat, 2 - kolo, 3 - trojkat, 0 - koniec
1
Podaj dlugosci bokow: 45
30
Pole figury wynosi 1350
Wykreslic figure? (t/n): n
Wybierz figure:
1 - prostokat, 2 - kolo, 3 - trojkat, 0 - koniec
2
Podaj promien: 40
Pole figury wynosi 5024
Wykreslic figure? (t/n): n
Wybierz figure:
1 - prostokat, 2 - kolo, 3 - trojkat, 0 - koniec
3
Podaj dlugosc podstawy i wysokosc: 350
250
Pole figury wynosi 43750
Wykreslic figure? (t/n): t
Wybierz figure:
1 - prostokat, 2 - kolo, 3 - trojkat, 0 - koniec
0
Koniec programu!
```

► Komputer ŚWIAT radzi

Etykieta jest po prostu specjalnie oznaczonym miejscem w kodzie. Aby wstawić etykietę, należy wpisać jej nazwę i dwukropek.

6 W zmiennej `pole` znajduje się obliczone przez nas pole figury. Wypisujemy więc tę wartość na ekranie ❼ i zadajemy użytkownikowi pytanie, czy narysować figurę ❽. Jeśli odpowiedź jest twierdząca, ponownie, wykorzystując instrukcję switch, wykreślamy odpowiednią figurę ❾ (wyświetlamy ją na środku okna o rozmiarach 400 na 400 pikseli).

7 Po skompilowaniu i uruchomieniu programu musimy solidnie przetestować napisany przez nas kod. Jeśli nie znajdziemy w nim żadnych błędów, możemy przystąpić do jego rozbudowy – dodać inne figury, umożliwić obliczanie obwodów. Pamiętajmy, aby co pewien czas przetestować stworzony fragment kodu.



▼ Funkcja int ()

Funkcja int () zamienia liczbę znajdującą się w nawiasach na liczbę całkowitą. Dzięki temu wartość zmiennej typu float możemy przypisać do zmiennej całkowitoliczbowej (oczywiście nie zostanie przypisana do niej część ułamkowa, lecz jedynie część całkowita).

5 Funkcje

Jeśli nasz program rozrósł się już do kilkudziesięciu linijek, warto pomyśleć o jego podziale na mniejsze części. Poznajmy więc funkcje

W rozdziale	strona
Co to jest funkcja	**83**
Definicja funkcji	83
Wywołanie funkcji	83
Definicja a deklaracja funkcji	84
Funkcja bezargumentowa	87
Funkcja nic niezwracająca	88
Funkcje na schemacie blokowym	89
Domyślne argumenty	89
Funkcjonalny program	90
Widoczność zmiennych	**92**
Zmienne lokalne	92
Zmienne globalne	93
Zmienne globalne zamiast argumentów	94
Przydatne funkcje matematyczne	**95**
Potęgowanie	95
Funkcje trygonometryczne	95
Pierwiastek	95
Logarytmy	95

Szybko się przekonamy, że funkcja to bardzo pożyteczny twór języka programowania. Pozwala wydzielić fragment programu wykonujący określone zadanie i w ten sposób uporządkować kod źródłowy.

Stosowanie funkcji ma też wiele innych zalet. Każdą funkcję możemy wywołać wielokrotnie w dowolnym miejscu naszego programu. Dzięki temu polecenia umieszczone wewnątrz funkcji w kodzie zapisywane są tylko raz, a wykonywane dowolną liczbę razy. Do funkcji możemy ponadto przesyłać różne dane i na ich podstawie wykonywać obliczenia, których wynik będzie zwracała funkcja.

Komputer ŚWIAT na koniec tego rozdziału omówi zagadnienie ściśle związane z funkcjami – widoczność zmiennych. Przekonamy się, że niektóre zmienne mogą być w określonych miejscach kodu niewidoczne, z innych będziemy mogli natomiast korzystać w każdej chwili.

CD-ROM

plik znajduje się na płycie dołączonej do książki

Nazwa na CD

plik w całości znajduje się na płycie dołączonej do książki, tu pokazany jest tylko jego fragment

Nazwa na CD

Co to jest funkcja

Poznane przez nas do tej pory konstrukcje pozwalały tworzyć programy liniowe. Dzięki funkcjom jeden fragment kodu będziemy mogli wykorzystać wielokrotnie

Do tej pory tworzyliśmy różne bloki instrukcji. Bloki te były wykonywane warunkowo (jeśli wykorzystaliśmy instrukcje warunkowe) lub wielokrotnie (gdy blok instrukcji stanowił ciało pętli). Poznaliśmy też instrukcję goto, dzięki której można w programie przejść w zupełnie inne miejsce kodu i wykonać znajdujące się tam instrukcje. Nadszedł czas, aby poznać dosyć specyficzny sposób na wyodrębnianie w jeden blok niektórych instrukcji – nauczmy się posługiwać funkcjami.

W poprzednich rozdziałach poznaliśmy już kilka funkcji. Już na pierwszych stronach tej książki wykorzystaliśmy funkcję o nazwie `main`, czyli tak zwaną funkcję główną. W ostatnim przykładzie znalazła się również inna funkcja – `int()`.

Definicja funkcji

Znajdujący się w każdym omawianym do tej pory przykładzie zapis ● (gdzie w miejscu ● znajdowały się różne instrukcje) to tak naprawdę definicja funkcji `main`. W definicji każdej funkcji muszą się znaleźć następujące elementy:

- określenie nazwy funkcji ❶ – funkcja główna musi się nazywać `main`,
- określenie typu danych będących wynikiem działania funkcji ❷ – ze strony 28 dowiedzieliśmy się, że funkcja `main` powinna zwracać liczbę całkowitą (a najlepiej liczbę 0 – wtedy dla systemu jest to informacja, że wykonanie funkcji głównej zostało zakończone sukcesem),
- argumenty (wraz z ich typami) funkcji ❸, czyli wartości przekazywane do wnętrza funkcji – do funkcji `main` przekazywane są dwa argumenty – na stronie 86 poznamy ich znaczenie,
- instrukcje wykonywane wewnątrz funkcji ❹ – dla funkcji głównej dotychczas instrukcje te stanowiły cały kod naszego programu,
- instrukcję `return`, dzięki której jako wynik działania funkcji zwracana jest wartość zmiennej lub wyrażenia znajdującego się po słowie kluczowym `return` – funkcja główna zwraca wartość `EXIT_SUCCESS`, która jest równoważna liczbie 0.

```
❷
● int main(int argc, char *argv[])
  ❶                          ❸
  {
  ●   ❹
      system("PAUSE");
      return EXIT_SUCCESS;
  }
```

Wywołanie funkcji

Wiemy już ogólnie, jak zdefiniować funkcję. Ale po co to robić? Na przykład po to, aby ją w dowolnym miejscu programu wywołać, przekazując do niej dowolny jeden lub więcej argumentów (oczywiście takiego typu, jaki ustaliliśmy w definicji). Funkcja `main` jest dosyć specyficznym rodzajem funkcji – nie trzeba jej wywoływać, bo od niej zaczyna się wykonywanie naszego programu. Stwórzmy więc własną, bardzo prostą funkcję – najpierw ją zdefiniujemy, a potem razem z Komputer ŚWIATEM wywołamy wewnątrz funkcji `main`.

1 Ustalamy, że nasza funkcja będzie nazywała się `dodawanie()`. Będziemy do niej przekazywać dwie liczby całkowite, a efektem działania funkcji będzie suma tych liczb (wynik będzie również liczbą całkowitą). Tuż przed definicją funkcji `main` wpisujemy typ zwracanego przez naszą funkcję wyniku (`int` – oznacza, jak pamiętamy, typ całkowitoliczbowy) i wpisujemy nazwę funkcji.

Jak nazywać funkcje

Przy nazywaniu funkcji stosujemy zasady podobne do tych, którymi należy się kierować przy nadawaniu nazw zmiennym – patrz strona 37.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int dodawanie(int a, int b)
{
    return a+b;
}

int main(int argc, char *argv[])
{
    cout << "10 + 25 = ";
    cout << dodawanie(10,25) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_40

2 W nawiasach określamy argumenty, które zostaną przekazane do funkcji. Ustaliliśmy, że będą to dwie liczby całkowite – wpisujemy więc (nasze zmienne wejściowe nazywamy w dowolny sposób – oczywiście zgodnie z zasadami nadawania nazwy zmiennym).

3 Teraz wystarczy już tylko wewnątrz nawiasów klamrowych umieścić instrukcje, które zostaną wykonane przez funkcję. Ponieważ nasza funkcja `dodawanie()` jest niezwykle prosta, wpisujemy od razu słowo kluczowe `return` i wyrażenie. Dzięki temu funkcja zwraca sumę przekazanych do niej dwóch liczb.

4 Aby się przekonać o tym, że nasza funkcja naprawdę działa, w funkcji głównej umieszczamy wywołanie, przekazując jako argumenty dwie liczby – 10 i 25. Po skompilowaniu i uruchomieniu programu na ekranie powinniśmy zobaczyć następujący widok.

```
C:\biblioteczka\program_40\program_40.exe
10 + 25 = 35
```

Komputer ŚWIAT radzi

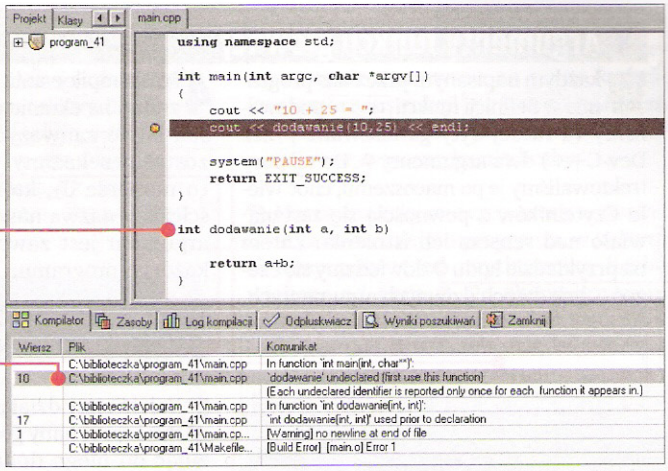
Oczywiście, do funkcji, zamiast liczb, możemy przekazać również wartość zmiennej. Przekazywane zmienne mogą mieć taką samą nazwę, jak w definicji naszej funkcji. Pamiętajmy jednak, że mimo iż są tak samo nazwane, to w rzeczywistości są to różne zmienne - na stronie 93 znajdziemy wyjaśnienie tego zjawiska. Nie ma również przeszkód, aby naszą funkcję wywołać w programie kilkakrotnie, przekazując do niej za każdym razem różne argumenty.

```
int x = 23;
int b = 7;

cout << x << " + " << b << " = ";
cout << dodawanie(x,b) << endl;
```

Definicja a deklaracja funkcji

W poprzednim przykładzie Komputer ŚWIAT umieścił definicję funkcji

`dodawanie()` tuż przed funkcją `main`. Sprawdźmy, co się stanie, jeśli zamienimy definicje tych funkcji miejscami. Próba skompilowania takiego programu kończy się wystąpieniem błędu. Dlaczego tak się dzieje? Odpowiedź jest prosta. Wywołanie funkcji `dodawanie()` wystąpiło przed jej definicją – to tak, jakbyśmy chcieli zapisać coś do zmiennej, której wcześniej nie zadeklarowaliśmy.



Czy więc skazani jesteśmy na definiowanie własnych funkcji przed funkcją główną? Na szczęście nie – byłoby to bardzo niewygodne, bo analizę programu należy zawsze rozpoczynać od funkcji głównej, która w takim wypadku znajdowałaby się gdzieś na końcu kodu. Jak więc rozwiązać ten problem? Naszą funkcję możemy umieścić poniżej funkcji głównej, ale przed pierwszym wywołaniem naszej funkcji należy ją zadeklarować.

Deklaracja funkcji od jej definicji różni się tym, że w deklaracji, zamiast wpisywania ciała funkcji, umieszczamy jedynie średnik. Deklaracja jest więc tylko zapowiedzią naszej funkcji. Kompilator na podstawie deklaracji wie, jaki typ wyniku zwraca funkcja oraz ile oraz jakiego typu argumenty należy do niej przekazać. Na tej podstawie może już sprawdzić, czy wywołanie naszej funkcji jest prawidłowe.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int dodawanie(int a, int b);

int main(int argc, char *argv[])
{
    cout << "10 + 25 = ";
    cout << dodawanie(10,25) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

int dodawanie(int a, int b)
{
    return a+b;
}
```

Program_41

► Komputer ŚWIAT radzi

Co prawda deklaracja funkcji może nastąpić tuż przed jej wywołaniem, jednak Komputer ŚWIAT poleca umieszczanie deklaracji funkcji przed funkcją główną. Dzięki temu kod jest czytelniejszy i łatwiej wprowadzać do niego zmiany.

Argumenty funkcji głównej

W każdym napisanym przez nas programie w definicji funkcji `main` podawaliśmy (a raczej były generowane przez Dev-C++) dwa argumenty. Do tej pory traktowaliśmy je po macoszemu, choć wielu Czytelników z pewnością się zastanawiało nad sensem ich istnienia. Zatem na przykładzie kodu dowiedzmy się czegoś o tych dwóch dziwnych argumentach.

```
int main(int argc, char *argv[])
{
    cout << "Liczba argumentow: " << argc << endl;
    int i = 0;
    for (int i=0; i < argc; i++)
    {
        cout << i << " -> " << argv[i] << endl;
    }
```

Program_42

1 W zmiennej całkowitoliczbowej `argc` znajduje się liczba przekazanych do funkcji `main` (czyli naszego programu) argumentów. Zmienna `*argv[]` to tablica (tę konstrukcję języka poznamy w następnym rozdziale), w której znajdują się te argumenty.

2 W linii wyświetlamy liczbę argumentów przekazanych do funkcji.

3 Następnie w pętli wyświetlamy kolejne argumenty (nie przejmujmy się tym, że nie rozumiemy tego zapisu – w następnym rozdziale wszystko się wyjaśni). Dla ułatwienia przed każdym argumentem wyświetlamy jego numer oraz strzałkę.

4 Po skompilowaniu i uruchomieniu programu na ekranie pojawia się widok. Jak łatwo zauważyć, do funkcji `main` został przekazany jeden argument (o numerze 0), którego wartością jest ścieżka i nazwa naszego programu. Ten argument jest zawsze dostarczany do każdego programu.

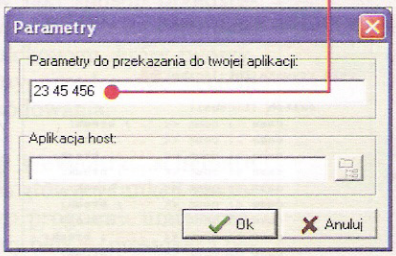
```
C:\biblioteczka\program_42\program_42.exe
Liczba argumentow: 1
0 -> C:\biblioteczka\program_42\program_42.exe
```

5 Zakończmy działanie naszego programu i spróbujmy go uruchomić, przekazując do niego dodatkowe argumenty. Najprościej uruchomić Wiersz polecenia (aby to zrobić, klikamy na Start, wybieramy Uruchom... i w pole Otwórz: wpisujemy `cmd`). Następnie w otwartym oknie przechodzimy do katalogu, w którym znajduje się nasz program (na przykład za pomocą polecenia) i uruchamiamy go, wpisując jego nazwę, a po spacji kilka wyrazów lub liczb oddzielonych od siebie odstępami. Po wciśnięciu klawisza enter na ekranie pojawia się widok. Widzimy więc, że aby do funkcji `main` przekazać argumenty, wystarczy podać je przy uruchamianiu programu.

```
C:\WINDOWS\System32\cmd.exe - program_42.exe 23 napis 245A

C:\Documents and Settings>cd c:\biblioteczka\program_42

C:\biblioteczka\program_42>program_42.exe 23 napis 245A
Liczba argumentow: 4
0 -> program_42.exe
1 -> 23
2 -> napis
3 -> 245A

Aby kontynuować, naciśnij dowolny klawisz . . .
```

► Komputer ŚWIAT radzi

Argumenty do funkcji głównej możemy przekazać również z poziomu aplikacji Dev-C++. Aby to zrobić, z menu Uruchom wybieramy pozycję Parametry... i w pole wpisujemy argumenty. Po kliknięciu na OK kompilujemy i uruchamiamy program – argumenty zostaną przekazane.



Funkcja bezargumentowa

Najprostszym przypadkiem funkcji jest jej wersja bezargumentowa. Spójrzmy na przykład.

1 Tworzymy deklarację i definicję funkcji o nazwie `pobierzLN()`. Funkcja będzie zwracała liczbę całkowitą. Do funkcji nie przekazujemy żadnych argumentów.

2 Działanie funkcji `pobierzLN()` polega na sprawdzeniu, czy wprowadzona przez użytkownika liczba jest większa od zera. Jeśli liczba będzie większa od zera, zostanie ona zwrócona przez funkcję (za pomocą instrukcji `return`), w innym wypadku zostanie wyświetlona prośba o ponowne wprowadzenie liczby.

3 Naszą funkcję możemy teraz wywołać w każdym miejscu programu, w którym chcemy pobrać od użytkownika liczbę większą od zera. Po skompilowaniu i uruchomieniu programu efekt jego działania jest taki.

```
C:\biblioteczka\program_43\program_43.exe
Podaj liczbe wieksza od 0: -6
Podaj liczbe wieksza od 0: 45
Zmienna a wynosi: 45
Podaj liczbe wieksza od 0: 0
Podaj liczbe wieksza od 0: 34
Zmienna wynosi: 34
```

Program_43

```
int pobierzLN();

int main(int argc, char *argv[])
{
    int a;
    a = pobierzLN();
    cout << "Zmienna a wynosi: " << a << endl;
    cout << "Zmienna wynosi: " << pobierzLN() << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

int pobierzLN()
{
    int liczba;
    while(1)
    {
        cout << "Podaj liczbe wieksza od 0: " ;
        cin >> liczba;
        if (liczba > 0) return liczba;
    }
}
```

Różna liczba argumentów

Wiemy, że możliwe jest stworzenie funkcji bezargumentowej. Widzieliśmy też kilka przykładów funkcji `dodawanie()`, która pobierała dwa argumenty (Komputer ŚWIAT proponuje eksperymenty z wywoływaniem funkcji z inną liczbą argumentów, niż została ona zdefiniowana). Funkcja `main` jest natomiast przykładem funkcji o zmiennej liczbie argumentów. Oczywiście możemy stworzyć własną funkcję o zmiennej liczbie argumentów. Mimo że z tej książki (ze względu na ograniczone miejsce) nie dowiemy się, jak to zrobić, warto wiedzieć, że istnieje taka możliwość.

Funkcja nic niezwracająca

To, że do funkcji nie musimy przekazywać żadnego argumentu, już wiemy. Warto jeszcze wiedzieć, że funkcja nie musi również niczego zwracać. Pojawia się tylko jedno pytanie: W jaki sposób zadeklarować taką funkcję? Do tego celu służy specjalny typ danych void, który oznacza... brak typu.

Komputer ŚWIAT radzi

Jeżeli ciekawi nas, co się stanie, gdy zadeklarujemy i zdefiniujemy funkcję bez podania żadnego typu danych zwracanego przez funkcję, sprawdźmy to.
Dowiemy się, że mimo braku określenia typu, funkcja będzie zwracała wartości całkowitoliczbowe - inaczej mówiąc, domyślnym typem dla funkcji jest podobnie jak dla zmiennych typ `int`.

1 Napiszmy funkcję, której zadaniem będzie wyświetlanie na ekranie rzymskiego zapisu przekazanej do funkcji liczby (dla uproszczenia zadania przyjmijmy, że funkcja będzie działać dla liczb od 1 do 10).

```
void rzymska(int liczba); (1)

int main(int argc, char *argv[])
{
    int x;
    cout << "Podaj liczbe od 1 do 10: ";
    cin >> x;
    cout << "Zapis rzymski liczby to: ";
    rzymska(x); (3)
    cout << endl << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

void rzymska(int liczba) (2)
{
     switch (liczba){
          case 1: cout << "I"; break;
          case 2: cout << "II"; break;
          case 3: cout << "III"; break;
          case 4: cout << "IV"; break;
          case 5: cout << "V"; break;
          case 6: cout << "VI"; break;
          case 7: cout << "VII"; break;
          case 8: cout << "VIII"; break;
          case 9: cout << "IX"; break;
          case 10: cout << "X"; break;
          default: cout << liczba;
     }
}
```

Program_44

2 Na początku zadeklarujmy naszą funkcję (1). Zwróćmy uwagę na typ, jaki przypisaliśmy zwracanej przez nią wartości. Słowo kluczowe void oznacza, że funkcja nie zwraca wyniku, więc nie musi się w niej pojawiać instrukcja `return`.

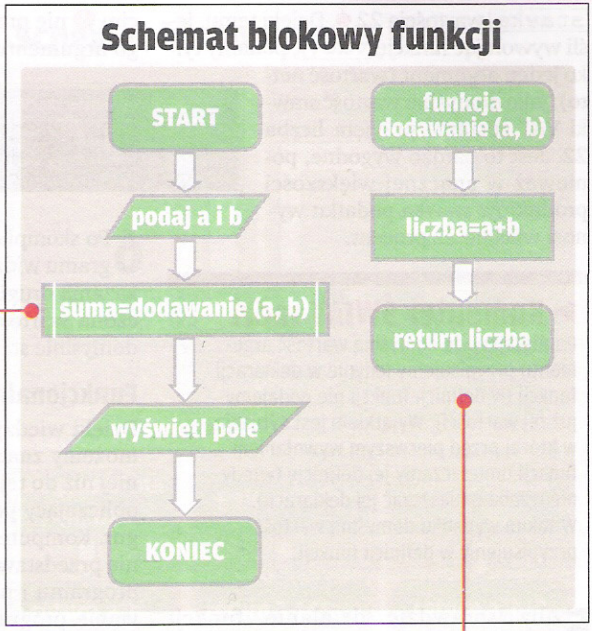
3 Poniżej funkcji `main` definiujemy naszą funkcję (2) – jej działanie powinno być jasne dla każdego.

4 Wewnątrz funkcji głównej możemy teraz wywoływać funkcję `rzymskie`, przekazując do niej liczbę całkowitą (3). Po skompilowaniu i uruchomieniu programu na ekranie pojawia się taki widok:

```
C:\biblioteczka\program_44\program_44.exe
Podaj liczbe od 1 do 10: 7
Zapis rzymski liczby to: VII
```

Funkcje na schemacie blokowym

Funkcje w naszym kodzie wprowadzają ład i upraszczają wszelkie modyfikacje programu. Niewątpliwie warto, a nawet trzeba z nich korzystać. Dlatego też należy wiedzieć, w jaki sposób przedstawiać funkcję na schemacie blokowym, który tworzymy przed rozpoczęciem kodowania.



1 Wewnątrz schematu głównej funkcji naszego programu umieszczamy blok (prostokąt z podwójnymi pionowymi bokami). W prostokąt wpisujemy nazwę funkcji. Możemy również wpisać, co do niej przekazujemy, a jeżeli wynik jej działania zapisujemy w zmiennej, również możemy umieścić taką informację.

2 Instrukcje znajdujące się wewnątrz naszej funkcji rozrysowujemy na innej kartce papieru (lub w pustym miejscu tej samej kartki), na przykład w taki sposób.

Domyślne argumenty

Twórcy języka C++ umożliwili przypisanie wartości domyślnych do jednego lub wielu argumentów przekazywanych do funkcji. Zobaczmy, jakie możemy dzięki temu osiągnąć korzyści (kod znajdziemy na następnej stronie).

1 Napiszmy definicję funkcji o nazwie `vat()`, która będzie zwracała wartość pierwszego argumentu powiększoną o liczbę procent przekazaną w drugim argumencie. Jako typ wyniku zwracanego przez funkcję wpisujemy więc `float`.

2 Pierwszy argument również powinien mieć wartość `float`, a drugi (określający stawkę procentową VAT-u) typ `int`.

3 Wewnątrz funkcji `vat()` umieszczamy stosowne obliczenia i zwracamy ich wynik za pomocą instrukcji `return`.

4 Ponieważ definicję funkcji `vat()` umieściliśmy na końcu naszego kodu, musimy ją zadeklarować przed pierwszym wywołaniem (najlepiej tuż przed definicją funkcji głównej). W deklaracji funkcji `vat()` inicjujemy zmienną

`stawka` wartością 22 ❶. Dzięki temu, jeśli wywołując funkcję `vat()`, podamy tylko jeden argument (wartość netto), jako domyślna wartość stawki VAT zostanie przyjęta liczba 22. Jest to bardzo wygodne, ponieważ w znacznej większości produktów stawka podatku wynosi właśnie 22 procent.

► Komputer ŚWIAT radzi

Pamiętajmy, że domyślną wartość argumentu przypisujemy jedynie w deklaracji funkcji (w definicji funkcji nie podajemy już tej wartości). Wyjątkiem jest sytuacja, w której przed pierwszym wywołaniem funkcji umieszczamy jej definicję (wtedy nie trzeba umieszczać jej deklaracji). W takim wypadku domyślną wartość przypisujemy w definicji funkcji.

5 Aby sprawdzić działanie funkcji `vat()`, wewnątrz funkcji `main` umieszczamy jej wywołania. W pierwszym ❶ jako drugi argument przekazujemy liczbę 22, w drugim ❷ – 7, a w trzecim ❸ nie przekazujemy w ogóle drugiego argumentu.

```
D:\aktualne\biblioteczka2\biblioteczka\program_45\program_45.exe
Cena=100 zl, VAT=22%, wartosc brutto wynosi 122
Cena=100 zl, VAT=7%, wartosc brutto wynosi 107
Cena=100 zl, VAT=22%, wartosc brutto wynosi 122
```

6 Po skompilowaniu i uruchomieniu programu widzimy, że w wypadku nieprzekazania drugiego argumentu została obliczona wartość dla ustawionej przez nas domyślnie stawki 22 procent ❶.

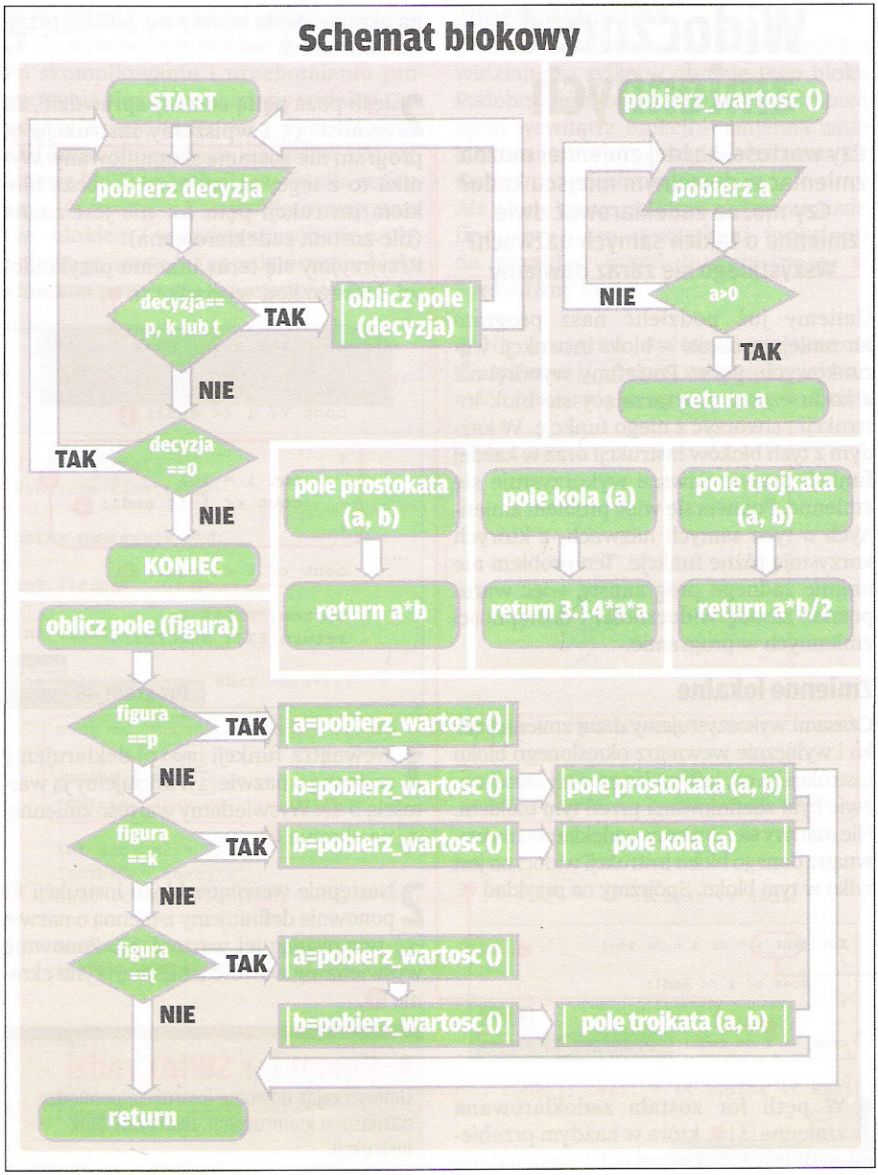
Funkcjonalny program

Dzięki wiedzy zdobytej w tym rozdziale możemy znacznie prościej i dużo czytelniej niż do tej pory napisać nasz program obliczający pola powierzchni różnych figur. Komputer ŚWIAT na następnej stronie przedstawia schemat blokowy takiego programu i poleca Czytelnikom zakodowanie programu samodzielnie. Oczywiście gotowy kod znajdziemy również na płycie dołączonej do tej książki (program znajduje się w katalogu program_46).

```
float vat(float x, int stawka = 22);

int main(int argc, char *argv[])
{
  ❶ cout << "Cena=100 zl, VAT=22%, wartosc brutto wynosi " << vat(100,22) << endl;
  ❷ cout << "Cena=100 zl, VAT=7%, wartosc brutto wynosi " << vat(100,7) << endl;
  ❸ cout << "Cena=100 zl, VAT=22%, wartosc brutto wynosi " << vat(100) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

float vat(float x, int stawka)
{
    return x * (1+float(stawka)/100);
}
```

Program_45

Schemat blokowy



Widoczność zmiennych

Czy wartości każdej zmiennej można zmieniać w dowolnym miejscu kodu? Czy można zadeklarować dwie zmienne o takich samych nazwach? Wszystkiego się zaraz dowiemy

Umiemy już podzielić nasz program na mniejsze części – bloki instrukcji warunkowych, pętle. Potrafimy wyodrębnić z kodu często powtarzający się blok instrukcji i stworzyć z niego funkcję. W każdym z tych bloków instrukcji oraz w każdej funkcji prawie zawsze wykorzystuje się zmienne. Pojawia się więc problem zmiennych o tych samych nazwach, z których korzystają różne funkcje. Ten problem nie ominie żadnego programisty, więc warto poznać zasadę widoczności i dostępności zmiennych w programie.

Zmienne lokalne

Czasami wykorzystujemy daną zmienną tylko i wyłącznie wewnątrz określonego bloku instrukcji. A co, jeśli zmienna o tej samej nazwie była zdefiniowana przed tym blokiem. Nie martwy się. Zmienna zadeklarowana wewnątrz danego bloku instrukcji widoczna jest tylko w tym bloku. Spójrzmy na przykład.

```
for (int i = 0; i < 5; i++)
{
    cout << i << endl;
}
cout << i << endl;
```

Program_47

1 W pętli for została zadeklarowana zmienna `i`, która w każdym przebiegu pętli jest zwiększana o 1 i wyświetlana na ekranie. Pętla kończy się, jeśli `i` przyjmie wartość 5.

2 Jeśli poza pętlą chcemy sprawdzić, ile wynosi `i`, i wpiszemy instrukcję, program nie zostanie skompilowany. Wynika to z tego, że zmienna `i` poza blokiem instrukcji pętli for nie jest znana (nie została zadeklarowana).

Przyjrzyjmy się teraz innemu przykładowi, którego kod wygląda tak.

```
int main(int argc, char *argv[])
{
    int i = 5; (1)
    cout << i << endl; (2)

    {
        int i = 10; (3)
        cout << i << endl; (4)
    }

    cout << i << endl; (5)

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Program_48

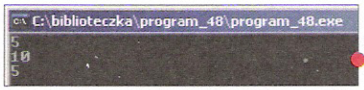
1 Wewnątrz funkcji `main` deklarujemy zmienną o nazwie `i` i inicjujemy ją wartością 5 (1). Wyświetlamy wartość zmiennej `i` na ekranie (2).

2 Następnie wewnątrz bloku instrukcji (3) ponownie definiujemy zmienną o nazwie `i`, przypisując jej wartość 10. Ponownie wyświetlamy wartość zmiennej `i` na ekranie (4).

► Komputer ŚWIAT radzi

Umieszczając dowolne instrukcje pomiędzy nawiasami klamrowymi, tworzymy blok instrukcji.

3 Po zakończeniu bloku instrukcji jeszcze raz wyświetlamy wartość zmiennej `i` 5. Po skompilowaniu i uruchomieniu programu na ekranie zobaczymy widok. Jak więc widzimy, zmienna `i` zadeklarowana (oraz zainicjowana wartością 10) wewnątrz bloku instrukcji przysłoniła zmienną `i` (o wartości 5) zadeklarowaną poza tym blokiem i w żaden sposób nie wpłynęła na jej wartość (po zakończeniu bloku zmienna ponownie ma wartość 5).



Zmienne globalne

Zmienne zdefiniowane w bloku instrukcji widziane są tylko w obrębie tego bloku. Podobnie jest ze zmiennymi zdefiniowanymi wewnątrz funkcji – zmienna zadeklarowana na jej początku widoczna jest wyłącznie w tej funkcji.

Ale co się stanie, jeśli naszą zmienną zdefiniujemy poza wszystkimi funkcjami, na przykład przed funkcją `main`. Sprawdźmy to.

1 Na początku wyświetlamy wartość zmiennej `liczba`, zmieniamy jej wartość i ponownie ją wyświetlamy. Dzięki

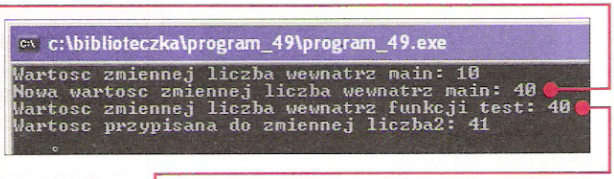
```
#include <cstdlib>
#include <iostream>

using namespace std;

int liczba = 10;

int test();

int main(int argc, char *argv[])
{
    cout << "Wartosc zmiennej liczba wewnatrz main: " << liczba << endl;

    liczba = 40;
    cout << "Nowa wartosc zmiennej liczba wewnatrz main: " << liczba << endl;

    int liczba2 = 0;
    liczba2 = test();
    cout << "Wartosc przypisana do zmiennej liczba2: " << liczba2 << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}

int test()
{
    cout << "Wartosc zmiennej liczba wewnatrz funkcji test: " << liczba << endl;
    return ++liczba;
}
```

Program_49

Zmienna lokalna

Zmienną dostępną tylko w określonym fragmencie kodu (na przykład zadeklarowaną wewnątrz pojedynczego bloku instrukcji i tylko w nim dostępną) nazywamy zmienną lokalną.

temu przekonamy się, czy zmienną `liczba` można wykorzystywać tak, jak zmienną zdefiniowaną wewnątrz funkcji `main`. Po skompilowaniu okaże się, że ze zmiennej zadeklarowanej przed funkcją `main` możemy normalnie korzystać ●.



2 Wewnątrz funkcji `main` wywołujemy teraz bezargumentową funkcję `test()`, której zadaniem będzie wyświetlenie wartości zmiennej `liczba` i zwrócenie tej wartości zwiększonej o 1. Jeśli taki kod zadziała, przekonamy się o tym, że zmienną `liczba` możemy wykorzystywać również wewnątrz innych funkcji. Jak się okaże po kompilacji i uruchomieniu, na ekranie zostanie wyświetlona prawidłowa wartość zmiennej `liczba` ●.

3 Na koniec wyświetlamy wartość zmiennej `liczba2`, do której przypisaliśmy wynik działania funkcji `test()` (która, jak pamiętamy, zwróciła zwiększoną o 1 wartość zmiennej `liczba`). Jak szybko zauważymy po skompilowaniu i uruchomieniu programu, ze zmiennej `liczba` można korzystać w każdej funkcji naszego programu – możemy ją bez problemów wyświetlać, zmieniać i przypisywać w każdym miejscu naszego kodu.

Zmienne globalne zamiast argumentów

Skoro zmienne globalne widziane są we wszystkich funkcjach naszego programu, moglibyśmy nimi zastąpić argumenty przekazywane do tych funkcji. No bo po co przekazywać jakąś wartość do funkcji, skoro można uczynić z niej wartość globalną i wykorzystywać w każdej funkcji. Niestety, w taki sposób myśli wielu początkujących programistów, nagminnie wykorzystując zmienne globalne. Kategorycznie powinniśmy się wystrzegać takich praktyk, ograniczając korzystanie ze zmiennych globalnych do minimum, czyli do sytuacji, w której zmienna globalna wykorzystywana jest przez większość funkcji w naszym kodzie, ale jej wartość nie jest przez nie zmieniana. Przykładem może być zmienna, której na początku działania programu przypisujemy wartość pobraną od użytkownika (na przykład jego imię) i później korzystamy z niej w różnych funkcjach naszego programu.

Zmienne globalne

Jeśli zmienna została zadeklarowana poza funkcją `main` (oraz poza wszystkimi innymi funkcjami) widoczna jest w całym naszym programie. Taką zmienną nazywamy zmienną globalną (w przeciwieństwie do zmiennych lokalnych widocznych tylko w określonych funkcjach czy blokach instrukcji).

Przydatne funkcje matematyczne

W programach komputerowych występuje zazwyczaj wiele obliczeń. Poznajmy kilka standardowych funkcji, które nam je ułatwią

Na stronie 39 poznaliśmy podstawowe operatory matematyczne. Za ich pomocą możemy wykonywać takie operacje, jak dodawanie, odejmowanie, mnożenie, dzielenie i dzielenie modulo. Niestety, w wielu wypadkach takie operacje nie będą nam wystarczały. Warto więc poznać kilka przydatnych funkcji matematycznych.

► Komputer ŚWIAT radzi

Definicje omawianych funkcji matematycznych znajdują się w dodatkowej bibliotece `cmath`. Musimy dołączyć ją do programu, w którym zamierzamy z tych funkcji korzystać. Aby to zrobić, wystarczy na początku kodu dopisać dyrektywę `#include <cmath>`.

Potęgowanie

Czasami zachodzi potrzeba podniesienia jednej liczby do określonej potęgi. Możemy do tego celu wykorzystać funkcję ●. Przy jej wywołaniu podajemy dwa argumenty – oba muszą być typu zmiennoprzecinkowego ● (jeśli przekazujemy do funkcji nie zmienną, lecz wartość liczbową, nawet jeśli jest całkowita, musimy postawić po niej kropkę i podać wartość ułamkową – na przykład 0 ●). Przydatna może się okazać również funkcja ●, która podnosi często wykorzystywaną liczbę e (równą około 2,72) do ustalonej przez nas za pomocą argmentu potęgi (w tym wypadku argument również musi być typu zmiennoprzecinkowego). Obie funkcje zwracają wartość zmiennoprzecinkową.

```
float z;
float x = 2, y = 10;

z = pow(x, y);
```

```
z = pow(2.0,10.0);
```

```
z = exp(3.0);
```

Funkcje trygonometryczne

Sinus, cosinus czy tangens – bez możliwości obliczania wartości tych funkcji nie obędziemy się w wielu przypadkach (związanych na przykład z wyświetlaniem grafiki). Jak z nich korzystać? Wystarczy wpisać skróconą nazwę funkcji trygonometrycznej ● i argument ● (typu zmiennoprzecinkowego).

```
z = sin(x);

z = cos(2.5);

z = tan(1.0);
```

Pierwiastek

Pierwiastek trudno obliczyć samodzielnie. Dlatego warto wiedzieć o istnieniu funkcji wyliczającej pierwiastek kwadratowy dowolnej liczby ● (musi mieć typ rzeczywisty).

```
z = sqrt(x);
```

Logarytmy

Umiemy już obliczać potęgi różnych liczb (w tym również liczby e). Warto jeszcze poznać funkcje logarytmujące. Za pomocą funkcji ● obliczymy logarytm naturalny podanej w argumencie liczby. Wykorzystując natomiast funkcję ●, obliczymy logarytm dziesiętny. W obu wypadkach argumenty muszą być typu zmiennoprzecinkowego.

```
z = log(x);
```

```
z = log10(100.0);
```

6 Tablice, wskaźniki i inne typy

Znamy już podstawowe typy danych. Najwyższa pora nauczyć się posługiwać bardziej zaawansowanymi strukturami języka C++

W rozdziale	strona
Niezwykłe tablice	**97**
Tablica jednowymiarowa	97
Tablice wielowymiarowe	98
Tablice bez wymiarów	99
Elementy poza tablicą	100
Tablica znakowa	101
Łańcuch znakowy	102
Lepsze łańcuchy znaków	103
Wskaźniki	**104**
Adres a wartość zmiennej	104
Zmienna wskaźnikowa	105
Zmiana wartości za pomocą wskaźnika	106
Wskaźniki a tablice	107
Wskaźnik do łańcucha znaków	108
Przekazywanie wskaźników do funkcji	**109**
Tablica jako argument funkcji	109
Wskaźnikowe argumenty	110

Każda poznana do tej pory zmienna może przechowywać jedną liczbę. Jeśli zaczniemy pisać bardziej rozbudowane programy, okaże się to niewystarczające. Warto więc poznać zmienne, które mogą przechowywać nawet kilkadziesiąt wartości.

Na początku dowiemy się, czym są tablice. Odkryjemy również, że za pomocą tablicy (oczywiście odpowiedniego typu) możemy stworzyć łańcuch znakowy.

W dalszej części rozdziału poznamy bardzo ciekawy typ danych – wskaźnikowy. Taka zmienna może przechowywać adres komórki (w pamięci naszego komputera), w której znajduje się wartość na przykład innej zmiennej. Choć zagadnienie to wydaje się skomplikowane, warto stawić mu czoło.

Na koniec omówione typy danych wykorzystamy w praktyce – tablice i wskaźniki przekażemy jako argumenty do stworzonej przez nas funkcji. Odkryjemy w ten sposób wiele ciekawych zależności.

CD-ROM

plik znajduje się na płycie dołączonej do książki

Nazwa na CD

plik w całości znajduje się na płycie dołączonej do książki, tu pokazany jest tylko jego fragment

Nazwa na CD

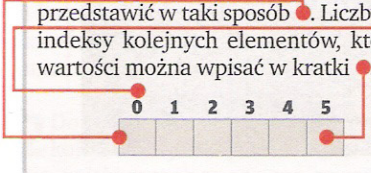
Niezwykłe tablice

Poznane typy danych pozwalają przechowywać pojedyncze liczby. Dzięki tablicom zgromadzimy wiele wartości w jednym miejscu

W każdym programie (no, może poza najprostszymi kilkulinijkowymi kodami) dużą rolę odgrywają zmienne. Na przykład, gdy chcieliśmy obliczyć pole prostokąta, prosiliśmy użytkownika o podanie dwóch liczb określających długości boków figury. Liczby te następnie zapisaliśmy do dwóch zmiennych o różnych nazwach (na przykład `a` i `b`). Zastanówmy się, ilu zmiennych potrzebowalibyśmy w programie obliczającym powierzchnię na przykład mieszkania? Z pewnością zbyt wielu. W jaki więc sposób zapisać dużo wartości w kodzie źródłowym? Najlepszym sposobem jest wykorzystanie do tego celu zmiennej tablicowej (określanej najczęściej jako tablica lub macierz). Poznajmy więc zasadę tworzenia tablic i korzystania z nich.

Definicja tablicy

Tablica jest zbiorem elementów jednego typu. Wszystkie te elementy umieszczane są w pamięci naszego komputera razem w jednym miejscu i do każdego z tych elementów możemy się odwołać, posługując się nazwą tablicy i indeksem określającym numer danego elementu. Najprostszą tablicę można więc przedstawić w taki sposób. Liczby to indeksy kolejnych elementów, których wartości można wpisać w kratki.



Numery elementów

Elementy tablicy numerowane są od 0. Tak więc w zadeklarowanej przez nas tablicy pięcioelementowej kolejne elementy są ponumerowane od 0 do 4. Indeks 0 odnosi się do pierwszego elementu, indeks 1 do drugiego i tak dalej, aż do indeksu 4, który reprezentuje piąty (ostatni) element tablicy `oceny`.

Tablica jednowymiarowa

Najprostszym przykładem tablicy jest jej jednowymiarowa postać. Spójrzmy na przykład wykorzystujący tego typu tablicę.

1 Wewnątrz funkcji `main` wpisujemy instrukcję 1. Należy ją rozumieć w następujący sposób: Stwórz zmienną tablicową o nazwie `oceny`, która składa się z pięciu elementów typu całkowitoliczbowego (`int`).

Program_50

```
int oceny[5]; 1

oceny[0] = 4;
oceny[1] = 5;
oceny[2] = 4; 2
oceny[3] = 3;
oceny[4] = 2;

for (int i=0; i<=4; i++) 3
{
    cout << "Przedmiot nr " << i;
    cout << " - ocena: " << oceny[i];
    cout << endl;
}
```

Komputer ŚWIAT radzi

Jak widzimy, tablicę deklaruje się podobnie jak każdą inną zmienną z tą różnicą, że w wypadku tablicy po jej nazwie w nawiasach kwadratowych wpisujemy rozmiar tablicy. Nawiasy kwadratowe mówią kompilatorowi, że ma do czynienia właśnie z tablicą.

Inicjacja tablicy

W artości tablicy możemy wpisać również podczas jej deklaracji. W tym celu wartości kolejnych elementów tablicy oddzielone od siebie przecinkami umieszczamy w nawiasach klamrowych – na przykład ●.

```
int oceny[5] = {4, 5, 4, 3, 2};
```

2 Wpiszemy teraz do tablicy kolejne wartości. Aby to zrobić, wystarczy podać nazwę tablicy, w nawiasach kwadratowych umieścić numer elementu, który chcemy wpisać, i za pomocą operatora przypisania nadać mu określoną wartość (takiego typu, jak ustaliliśmy w deklaracji zmiennej tablicowej) ❷.

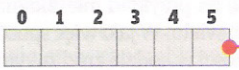
3 Tablica `oceny` powinna zostać wypełniona wartościami. Wyświetlimy je na ekranie, aby sprawdzić, czy rzeczywiście tak się stało. Do wyświetlenia najlepiej użyć pętli for. Tworzymy więc pętlę ❸ (w której zmienna iteracyjna przybiera wartości od 0 do 4 – zgodnie z indeksami naszej tablicy, więc możemy ją wykorzystać do indeksowania tablicy `oceny`).

4 Po skompilowaniu i uruchomieniu programu na ekranie widzimy ●.

```
C:\biblioteczka\program_50\program_50.exe
Przedmiot nr 0 - ocena: 4
Przedmiot nr 1 - ocena: 5
Przedmiot nr 2 - ocena: 4
Przedmiot nr 3 - ocena: 3
Przedmiot nr 4 - ocena: 2
```

Tablice wielowymiarowe

Poznane przez nas do tej pory tablice były jednowymiarowe – można je było przedstawić w postaci ponumerowanych pól ●.



W języku C++ (a także w prawie każdym języku programowania) można tworzyć tablice znacznie bardziej złożone – jak na przykład tablica dwuwymiarowa, którą możemy graficznie zobrazować w taki spo-

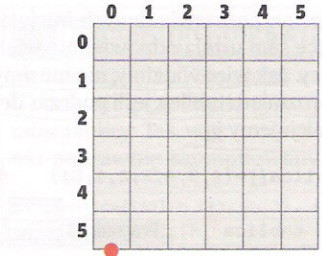
Dane w pętli

Pętle (najczęściej typu for) wykorzystuje się często zarówno do wpisywania do tablicy wartości, jak też do wyprowadzania ich na ekran. Spójrzmy na przykład ●, w którym do tablicy `potegi_dwojki` wpisujemy 10 kolejnych potęg liczby 2. Wykorzystując drugą pętlę, wyświetlamy wartości tablicy na ekranie ●.

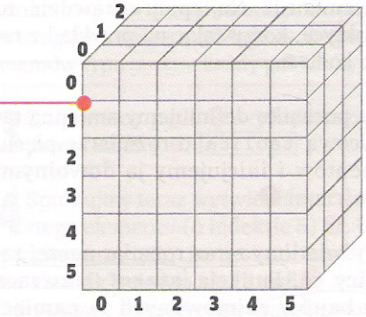
Program_51

```
int potegi_dwojki[10];
for (int j=0; j<10; j++)
{
    int wynik = 1;
    for (int i=0; i<j; i++) wynik*=2;
    potegi_dwojki[j] = wynik;
}

for(int i = 0; i < 10; i++)
{
        cout << "2 do potegi " << i << " wynosi ";
        cout << potegi_dwojki[i] << endl;
}
```

```
C:\biblioteczka\program_51\program_51.exe
2 do potegi 0 wynosi 1
2 do potegi 1 wynosi 2
2 do potegi 2 wynosi 4
2 do potegi 3 wynosi 8
2 do potegi 4 wynosi 16
2 do potegi 5 wynosi 32
2 do potegi 6 wynosi 64
2 do potegi 7 wynosi 128
2 do potegi 8 wynosi 256
2 do potegi 9 wynosi 512
```



sób (czyli jako tabelę składającą się z określonej liczby kolumn i wierszy). Bez problemów stworzymy tablicę trójwymiarową (na przykład taką) oraz tablicę o większej liczbie wymiarów. Jednak ze względu na to, że tablice o liczbie wymiarów większych od 3 trudno sobie nawet wyobrazić, rzadko się je stosuje. Zobaczmy, jak zadeklarować tablicę dwuwymiarową i jak z niej korzystać.



1 Tablicę dwuwymiarową deklarujemy podobnie jak jednowymiarową. Jedyną różnicą jest dodatkowa para nawiasów kwadratowych. Rozmiar tablicy dwuwymiarowej określamy jako liczbę wierszy i kolumn (podając te liczby w nawiasach kwadratowych). Nasza tablica ma mieć wymiar 3 na 4.

2 Do tablicy w podwójnej pętli wpiszemy kolejne cyfry. Dzięki temu przekonamy się, w jaki sposób są one zapisywane w tablicy. Tworzymy więc pętlę, która będzie przechodziła przez kolejne wiersze tablicy, oraz pętlę przechodzącą przez kolejne kolumny tablicy w ramach danego wiersza. Do tablicy wpisujemy wartość zmiennej `liczba`, która jest w każdym przebiegu pętli zwiększana o 1.

Program_52

```
int tablica[3][4];
int liczba=1;

for (int i=0; i<3; i++)
{
    for (int j=0; j<4; j++)
    {
        tablica[i][j]=liczba;
        liczba++;
    }
}

for (int i=0; i<3; i++)
{
    for (int j=0; j<4; j++)
    {
        cout << "tablica[" << i << "][" << j << "]";
        cout << "=" << tablica[i][j] << endl;
        liczba++;
    }
}
```

3 Tablicę wyświetlamy w analogiczny sposób. Po skompilowaniu oraz uruchomieniu aplikacji na ekranie widzimy:

```
C:\biblioteczka\program_52\program_52.exe
tablica[0][0]=1
tablica[0][1]=2
tablica[0][2]=3
tablica[0][3]=4
tablica[1][0]=5
tablica[1][1]=6
tablica[1][2]=7
tablica[1][3]=8
tablica[2][0]=9
tablica[2][1]=10
tablica[2][2]=11
tablica[2][3]=12
```

Tablice bez wymiarów

Aby zadeklarować tablicę, musimy podać typ elementów, które będą się w niej znajdować, nazwę tablicy oraz jej rozmiar. Gdy nie podamy żadnego typu, kompilator jako domyślny typ ustawi `int`. A co się stanie, jeśli nie podamy rozmiaru tabli-

Inicjacja dwuwymiarowa

Podobnie, jak w wypadku tablicy jednowymiarowej, przy okazji deklaracji tablicy dwuwymiarowej możemy od razu wpisać do niej wartości. Są dwa sposoby zainicjowania tablicy dwuwymiarowej:

- możemy w nawiasach klamrowych podać kolejne wartości, oddzielając je od siebie przecinkami (liczba elementów w tablicy równa jest iloczynowi zadeklarowanych przez nas wierszy i kolumn),
- możemy podać wartości, obejmując kolejne wiersze dodatkowymi nawiasami klamrowymi – taki zapis jest bardziej czytelny.

```
tablica[3][4]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

tablica[3][4]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

cy? Sprawdźmy to, wpisując w kodzie linijkę. Próba kompilacji takiego kodu kończy się niepowodzeniem – pojawia się błąd treści: „storage size of 'tablica' isn't known" (ang. rozmiar tablicy nie jest znany).

```
int tablica[];
```

Spróbujmy jednak od razu przy deklaracji zainicjować tablicę, czyli przypisać wartości elementom tablicy. Taki program kompiluje się bez problemu, a próba wyświetlenia dowolnego elementu tablicy kończy się sukcesem. Dlaczego tak się dzieje?

Kompilator na podstawie danych inicjujących tablicę sam ustala odpowiednią wielkość tablicy. Jak więc widzimy, nie musimy podawać rozmiaru tablicy, jeśli podczas deklaracji inicjujemy ją.

```
int tablica[]={6,5,4,3,2,1,0};

cout << tablica [4];
```

Program_53

```
C:\biblioteczka\program_53\program_53.exe
2
```

Elementy poza tablicą

Do tej pory wpisywaliśmy do tablicy tylko tyle elementów, ile dla niej zadeklarowaliśmy. Czy nie zastanawia nas, co się stanie, jeśli na przykład spróbujemy do tablicy pięcioelementowej wpisać wartość szóstego elementu? Najlepiej sprawdzić to w praktyce, korzystając na przykład z takiego kodu.

1 Na początku definiujemy zmienną tablicową `tablica` o rozmiarze pięciu elementów i inicjujemy ją dowolnymi wartościami (1).

2 Wyświetlimy teraz rozmiar naszej tablicy (2). Funkcja `sizeof()` zwraca ilość bajtów zajmowanych w pamięci przez zmienną o przekazanej do niej nazwie. Tak więc na ekranie widzimy, że

Program_54

```
int tablica[5] = {5,4,3,2,1};                                          (1)
cout << "Rozmiar tablicy: " << sizeof(tablica) << endl;                (2)

for (int i=1; i<5; i++) cout << i << " -> " << tablica[i] << endl;     (3)

cout << endl << "5 -> " << tablica[5] << endl;                         (4)

tablica[5] = 20;                                                       (5)
cout << endl << "5 -> " << tablica[5] << endl;                         (6)

cout << "Rozmiar tablicy: " << sizeof(tablica) << endl;
```

Bezwymiarowa tablica wielowymiarowa

Pamiętajmy, że kompilator może samodzielnie przydzielić tylko jeden wymiar tablicy. Tak więc zapis nie zostanie poprawnie skompilowany. Bez wymiaru możemy pozostawić tylko jeden i do tego wyłącznie pierwszy wymiar tablicy. Zapis jest więc jak najbardziej poprawny.

```
int tablica[][3] = {{1, 2, 3}, {4, 5, 6}};
```

```
int tablica[][] = {{1, 2, 3}, {4, 5, 6}};
```

```
C:\biblioteczka\program_54\program_54.exe
Rozmiar tablicy: 20
1 -> 4
2 -> 3
3 -> 2
4 -> 1

5 -> 2293672

5 -> 20
Rozmiar tablicy: 20
```

zmienna `tablica` zajmuje w pamięci naszego komputera 20 bajtów (pięć elementów typu `int` – każdy po 4 bajty).

3 Za pomocą pętli (3) wyświetlamy elementy tablicy.

4 Spróbujmy teraz wyświetlić wartość szóstego elementu (o indeksie 5) (4). Łatwo przekonamy się, że jest to możliwe – na ekranie pojawia się dowolna liczba. Jej wartość będzie odpowiadała wartości komórki pamięci, która znajduje się tuż za naszą tablicą.

5 Wiemy, że możemy wyświetlić wartość elementu tablicy, który tak naprawdę się w niej nie znajduje (leży po prostu w pamięci komputera tuż za tablicą). Czy możemy coś zapisać do tego elementu? Warto spróbować (5). Szybko okaże się, że jest to możliwe. Po ponownym wyświetleniu tego elementu (6) widzimy, że wartość została normalnie przypisana.

6 Na koniec sprawdzamy jeszcze rozmiar naszej tablicy – widzimy, że mimo przypisania wartości dodatkowemu elementowi rozmiar tablicy się nie zmienia.

Komputer ŚWIAT radzi

Tablica ma zawsze taki wymiar (zostaje jej przydzielone tyle miejsca w pamięci), jaki ustalimy podczas jej deklaracji. Oczywiście można czytać i zapisywać wartości w elementach o indeksach większych niż ostatni indeks tablicy, ale nie należy tego robić. Dlaczego? Ponieważ możemy nadpisać obszar pamięci nienależący do naszej tablicy, ale przydzielony dla innej zmiennej, czy też dla innej aplikacji.

Tablica znakowa

Do tej pory posługiwaliśmy się najczęściej tablicami typu `int`. Oczywiście nasza tablica może być innego typu (może na przykład przechowywać liczby zmiennoprzecinkowe) – dla działania takiej tablicy nie ma to żadnego znaczenia. Jedynym wyjątkiem jest typ znakowy (`char`), dzięki któremu możemy tworzyć dosyć specyficzne tablice. Wyobraźmy sobie bowiem, że jako kolejne elementy tablicy zapisujemy znaki: K, o, m, p, u, t, e, r. Jeśli wyświetlimy później w pętli kolejne elementy tablicy, na ekranie zobaczymy napis Komputer. Tablica typu znakowego umożliwia nam więc

```
char napis[] = {'K','o','m','p','u','t','e','r',' ','S','W','I','A','T','\0'};

for (int i=0; napis[i]!='\0'; i++) cout << napis[i];
```

Program_55

przechowywanie ciągów znakowych, czyli napisów. Spójrzmy na przykład.

1 Deklarujemy tablicę typu znakowego o nazwie `napis` i inicjujemy ją wartościami. Jako ostatnią wartość umieszczamy znak zerowy – symbol \0 (więcej informacji w ramce Znak zerowy \0). Rozmiar tablicy `napis` kompilator automatycznie ustawia na 15 elementów.

Znak zerowy \0

Znak zerowy reprezentowany przez symbol \0 jest specjalnym znakiem służącym do oznaczania końca łańcucha tekstowego. Dzięki niemu tablica znakowa może być traktowana jako łańcuch znakowy, czyli niejako oddzielny typ danych przechowujących ciągi znaków.

2 W pętli typu for wyświetlamy kolejne elementy tablicy `napis`. Zwróćmy uwagę na warunek przejścia do kolejnego przebiegu pętli. Można go odczytać następująco: Wykonuj pętlę, dopóki wartość kolejnego elementu tablicy `napis` nie będzie symbolem \0. Po skompilowaniu i uruchomieniu programu na ekranie widzimy.

```
C:\biblioteczka\program_55\program_55.exe
Komputer SWIAT
```

Łańcuch znakowy

Oczywiście posługiwanie się łańcuchami znakowymi w przedstawiony przed chwilą przez Komputer ŚWIAT sposób byłoby bardzo niewygodne (a szczególnie pamiętanie o znaku zerowym). Na szczęście istnieją prostsze metody zapisania łańcucha znakowego. Spójrzmy na przykład.

Program_56

```
char napis[]="Dzien dobry"; (1)
char imie[10]; (2)

cout << "Podaj swoje imie: ";
cin >> imie; (3)

cout << napis << " " << imie; (4)
```

1 Na początku deklarujemy tablicę bezwymiarową (oczywiście typu znakowego) i inicjujemy ją łańcuchem znaków zapisanym w cudzysłowach (1). Taki zapis jest znacznie wygodniejszy niż wpisywanie jako elementów tablicy kolejnych znaków. Tablicy (możemy ją zwać łańcuchem znaków) o nazwie `napis` zostaje przydzielony rozmiar 12 (11 znaków plus znak zerowy na końcu).

2 Tworzymy jeszcze jedną tablicę znakową – o nazwie `imie` i rozmiarze 10 znaków (2). Zapisujemy w niej imię, które użytkownik wpisuje z klawiatury (3).

3 Teraz możemy wyświetlić oba łańcuchy znaków (4). W tym celu podajemy nazwę zmiennej tablicowej (bez nawiasów kwadratowych). Efekt działania programu może być na przykład taki.

```
C:\biblioteczka\program_56\program_56.exe
Podaj swoje imie: Pawel
Dzien dobry Pawel
```

Długość łańcucha

W ostatnim przykładzie zadeklarowaliśmy rozmiar zmiennej tablicowej `imie` na 10 znaków (a właściwie 9 znaków i znak zerowy). Co się stanie, jeśli użytkownik poda wyraz znacznie dłuższy od przewidzianego przez nas (na przykład taki)? Dziesięć pierwszych znaków tego napisu zostanie zapisanych w pamięci zarezerwowanej dla tablicy `imie`. Pozostałe znaki zapiszą się w kolejnych komórkach pamięci, nadpisując znajdujące się tam dane. W tym wypadku część znaków została wpisana w obszarze pamięci przeznaczonym dla tablicy `napis`. Dlatego zamiast ciągu „Dzien Dobry" na ekranie pojawia się fragment podanego przez użytkownika napisu . Pamiętajmy więc, aby deklarować rozmiar tablicy na tyle duży, aby pomieścił planowany łańcuch znaków. Nie warto jednak tworzyć bardzo dużych tablic – niepotrzebnie zajmują miejsce w pamięci.

```
C:\biblioteczka\program_56\program_56.exe
Podaj swoje imie: Bardzodługieiniezostaniezapisanepozatablica
zostaniezapisanepozatablica Bardzodługieiniezostaniezapisanepozatablica
```

Lepsze łańcuchy znaków

Łańcuchy znaków w postaci tablicy są niezbyt wygodne. Dlatego stworzono specjalny typ danych służący do reprezentacji napisów. Typ ten nazwano string – Komputer ŚWIAT wspomniał o nim przy okazji omawiania stałych na stronie 34. String ma wiele zalet w porównaniu z tablicą znakową. Warto wymienić te najważniejsze:

- używając typu string, nie musimy martwić się o przydział pamięci (zadeklarowanie tablicy o odpowiedniej wielkości),
- zmienna typu string zajmuje mniej miejsca niż tablica znaków, której rozmiar powinien być na tyle duży, aby zmieścił się w niej każdy napis, jaki przewidujemy w niej zapisać,
- wykorzystując string, mamy możliwość skorzystania z wielu pomocnych funkcji – wyszukujących, łączenia różnych napisów, porównywania dwóch napisów ze sobą.

Komputer ŚWIAT radzi

Definicja typu danych string znajduje się w jednym z plików dołączanych zawsze na początku naszego kodu za pomocą polecenia `#include <iostream>`.

Aby pokazać działanie typu string, Komputer ŚWIAT przygotował krótki kod ,

```
C:\biblioteczka\program_57\program_57.exe
Podaj imie: Jarek
Dzien dobry Jarek
Dzien dobry Jarek!
```

po skompilowaniu którego na ekranie widzimy . Komputer ŚWIAT zachęca do przeanalizowania tego kodu i eksperymentowania. Warto zwrócić szczególną uwagę na zapis , który pokazuje możliwość łączenia (za pomocą zwykłego operatora sumowania) wielu zmiennych typu string w jeden łańcuch znakowy.

```
string napis = "Dzien dobry ";
string imie;

cout << "Podaj imie: ";
cin >> imie;

cout << napis << imie << endl;

napis += imie + "!";
cout << napis << endl;
```

Program_57

Wskaźniki

Niektóre typy danych są dosyć dziwne. Przykładem jest typ wskaźnikowy – różni się znacznie od pozostałych typów i ma ciekawe własności

Zmienna wskaźnikowa jest dosyć ciekawym typem danych używanym przez początkujących sporadycznie (czasami nawet nie wiedzą oni, że korzystają ze wskaźnika – ale o tym później). Jednak bardziej zaawansowany programista bez wskaźników nie może się obyć. Jaka jest funkcja wskaźników (po ang. pointers)? Zgodnie z nazwą – wskazują miejsce, w którym znajduje się jakiś obiekt (stała, zmienna, tablica czy funkcja). Innymi słowy, zmienna wskaźnikowa przechowuje adres komórki pamięci naszego komputera, w której może znajdować się na przykład wartość innej zmiennej.

Z pewnością wielu Czytelników zastanawia się nad sensem uczenia się takiej, na pierwszy rzut oka, skomplikowanej i niepotrzebnej konstrukcji. Oto kilka najważniejszych zastosowań wskaźników:

- znajomość adresu danych w pamięci komputera i dzięki temu odwołanie się do nich za pomocą tego adresu, to w wielu wypadkach najszybszy sposób na dotarcie do tych danych,
- możemy wywoływanym funkcjom przekazywać adres w pamięci, pod którym przechowywane są wartości zmiennych czy innych (czasem bardzo dużych) struktur danych. Dzięki temu funkcje mogą operować na tych zmiennych (a nie tylko ich wartościach), a w wypadku przekazania wskaźnika do dużych struktur (na przykład tablic), program zajmuje mniej pamięci i wykonuje się znacznie szybciej.

Adres a wartość zmiennej

Zanim Komputer ŚWIAT pokaże, w jaki sposób używać wskaźników, przypomnijmy sobie zasadę działania zmiennej. Dzięki temu łatwiej będzie nam zrozumieć istotę wskaźników. Spójrzmy na kod.

Program_58

```
int x = 11;

cout << "Wartosc prawostronna: " << x << endl;
cout << "Wartosc lewostronna: " << &x;
```

1 Po zdefiniowaniu (zadeklarowaniu i zainicjowaniu) zmienna `x` ma dwie wartości: lewostronną i prawostronną. Prawostronną wartością (zwaną po prostu wartością) jest liczba 11. Wartość lewostronna to adres komórki w pamięci, w której przechowywana jest wartość prawostronna. Tak więc nazwa zmiennej (w tym wypadku `x`) jest wskazaniem miejsca w pamięci, w której należy szukać wartości zmiennej. Miejsce w pamięci przydzielane jest na stałe w momencie zadeklarowania zmiennej.

2 Wartość prawostronną (dla uproszczenia będziemy posługiwać się samym zwrotem wartość zmiennej) łatwo sprawdzić – wystarczy wyświetlić ją na ekranie za pomocą na przykład polecenia. Ale jak sprawdzić wartość lewostronną (dla uproszczenia będziemy posługiwać się określeniem adres zmiennej)? Również możemy go wyświetlić, wykorzystując do tego operator adresowy &.

3 Po skompilowaniu i uruchomieniu naszego kodu na ekranie zobaczymy widok.

```
C:\biblioteczka\program_58\program_58.exe
Wartosc prawostronna: 11
Wartosc lewostronna: 0x22ff74
```

Program_59

```
int x = 11;

cout << "Wartosc zmiennej x: " << x << endl;
cout << "Adres zmiennej x: " << &x << endl;
cout << "Wartosc zapisana w pamieci pod tym adresem: " << * &x;
```

Wiemy już, jak uzyskać adres zmiennej. Potrafimy go nawet wyświetlić na ekranie. Niestety, ta umiejętność raczej się w dalszej nauce programowania nie przyda. Dlatego zobaczmy, co możemy z uzyskanym adresem zrobić pożytecznego. Kod od poprzedniego różni się jedynie linijką. Wyświetlamy w niej wartość zapisaną pod adresem uzyskanym za pomocą operatora &. Używamy do tego operatora *. Jak zobaczymy po skompilowaniu i uruchomieniu kodu, operator ten spowoduje, że zamiast adresu pamięci zostanie wyświetlona wartość pod nim zapisana. Działanie operatora * jest więc dokładnie przeciwne do działania operatora &.

```
c:\biblioteczka\program_59\program_59.exe
Wartosc zmiennej x: 11
Adres zmiennej x: 0x22ff74
Wartosc zapisana w pamieci pod tym adresem: 11
```

Zmienna wskaźnikowa

Przedstawiony w poprzednich dwóch kodach zapis `&x` to nic innego, jak przykładowa wartość zmiennej wskaźnikowej (zmiennej typu wskaźnikowego). Czas dowiedzieć się, w jaki sposób zadeklarować zmienną, aby taką wartość można było jej przypisać.

Program_60

```
int x = 11;
int *px;
px = &x;

cout << "Wartosc zmiennej x: " << x << endl;
cout << "Wartosc wskaznika px: " << px << endl;
cout << "Wartosc zapisana w pamieci wskazywanej przez wskaznik px: " << *px;
```

1 Oprócz zmiennej całkowitoliczbowej `x` deklarujemy również zmienną wskaźnikową `px`, która będzie wskazywała na typ całkowitoliczbowy. Zwróćmy uwagę na operator *, który poprzedza nazwę zmiennej wskaźnikowej – to właśnie ten symbol informuje kompilator, że zmienna (w tym wypadku `px`) jest wskaźnikiem.

► Komputer ŚWIAT radzi

W różnych kodach możemy spotkać się z różnym formatem zapisu deklaracji zmiennej wskaźnikowej. Wszystkie są poprawne, jednak Komputer ŚWIAT proponuje trzymać się jednej formy zapisu - najlepiej.

```
int *px;

int* px;

int * px;
```

2 Skoro mamy już zmienną wskaźnikową, możemy jej przypisać adres zmiennej `x`. Z poprzednich przykładów wiemy już, jak uzyskać taki adres. Wystarczy więc użyć operatora przypisania (symbolizowanego przez znak =).

3 Tak jak poprzednio wyświetlamy wartość zmiennej `x` oraz wartość zmiennej wskaźnikowej `px` (będzie to adres zmiennej `x`). Na końcu wyświetlamy również wartość wskazywaną przez adres będący wartością zmiennej wskaźnikowej `px`. Mówiąc prościej, wyświetlamy daną zapisaną w komórce pamięci o adresie przechowywanym w zmiennej wskaźnikowej `px`.

Wartość lewo- i prawostronna wskaźnika

Zmienna wskaźnikowa też ma swoją wartość lewo- i prawostronną. Jednak w tym wypadku obie wartości są adresami w pamięci komputera – lewostronna to adres wskaźnika, który służy do odwoływania się do samego wskaźnika, a prawostronna to adres dowolnej innej komórki pamięci.

1 Deklarujemy zmienną `x` typu float oraz wskaźnik `px` tego samego typu ❶.

2 Zmiennej `x` przypisujemy dowolną wartość, a wskaźnikowi `px` adres zmiennej `x` ❷.

3 Wyświetlamy na ekranie wartość zmiennej `x`, adres zapisany w zmiennej wskaźnikowej `px` oraz wartość znajdującą się pod tym adresem w pamięci komputera ❸.

4 Po kompilacji i uruchomieniu programu na ekranie zobaczymy widok ●. Widzimy więc, że wskaźnik przechowuje adres pamięci, a odwołanie się do informacji zapisanej pod tym adresem uzyskamy za pomocą wyłuskania (czyli wykorzystania operatora *).

Zmiana wartości za pomocą wskaźnika

Korzystając ze wskaźników, możemy nie tylko odczytywać dane znajdujące się pod wskazywanym przez nie adresem pamięci, ale także modyfikować te dane. W jaki sposób to robić? Dokładnie tak samo, jak w wypadku zwykłych zmiennych. Spójrzmy na kod ●.

Program_61

```
float x, *px; ❶

x = 7.456;
px = &x; ❷

cout << "Wartosc x: " << x << endl;
cout << "Wartosc px: " << px << endl;
cout << "Wartosc wskazywana przez px: " << *px << endl; ❸

*px = 2.34 + 1; ❹

cout << "Wartosc x: " << x << endl;
cout << "Wartosc px: " << px << endl;
cout << "Wartosc wskazywana przez px: " << *px << endl;
```

Dereferencja, czyli wyłuskanie

Operację, którą wykonuje operator &, można określić mianem pobierania adresu. Operację przeciwną, czyli taką, którą wykonuje operator *, nazywamy dereferencją albo prościej wyłuskaniem (wyłuskujemy pewną wartość z obszaru pamięci). Warto przyswoić sobie te nazwy.

4 W linii ❹ zmieniamy wartość komórki pamięci znajdującej się pod adresem przechowywanym we wskaźniku `px`. Oczywiście wykorzystujemy do tego operator wyłuskania oznaczany symbolem gwiazdki *. Gdybyśmy go nie wpisali, nie zmienilibyśmy wartości zapisanej pod adresem, tylko zmodyfikowalibyśmy sam adres, który zapisany jest we wskaźniku o nazwie `px`.

Program_62

```
int tablica[5]={2, 20, 200, 2000, 20000};
int *ptablica1, *ptablica2;

ptablica1 = &tablica[0];
ptablica2 = tablica;

cout << "Wartosc wskaznika ptablica1: " << ptablica1 << endl;
cout << "Wartosc wskaznika ptablica2: " << ptablica2 << endl;
cout << "Wartosc dereferencji wskaźnika ptablica2: " << *ptablica2 << endl;
cout << "Wartosc wskaznika (ptablica2+1): " << ptablica2 + 1 << endl;
cout << "Wartosc dereferencji wskaźnika (ptablica1+1): " << *(ptablica2 + 1) << endl;
cout << "Wartosc dereferencji wskaźnika (ptablica1+2): " << *(ptablica2 + 2) << endl;
```

Wskaźniki a tablice

Wskaźniki i tablice są ze sobą w języku C++ ściśle związane. Aby się o tym przekonać, przyjrzyjmy się następnemu kodowi.

1 Zdefiniujmy tablicę pięcioelementową, przechowującą wartości całkowitoliczbowe. Deklarujemy również dwa wskaźniki `ptablica1` i `ptablica2` (oczywiście tego samego typu co tablica).

2 Do wskaźnika `ptablica1` przypisujemy adres pierwszego elementu tablicy (wykorzystując operator adresu &).

3 Spójrzmy na następną linię, w której do wskaźnika `ptablica2` przypisujemy zmienną tablicową `tablica`, nie wykorzystując operatora &. Wyświetlamy oba wskaźniki – po skompilowaniu i uruchomieniu programu widać, że oba zawierają identyczny adres (wyjaśnienie w ramce na następnej stronie).

4 Sprawdźmy wartość zapisaną w pamięci pod adresem przechowywanym we wskaźniku do tablicy. Na ekranie wyświetlona zostaje wartość pierwszego elementu tablicy.

5 Wiemy już, że wskaźnik do tablicy wskazuje na pierwszy element tablicy.

Mnożenie a wyłuskanie

Zwróćmy uwagę, że operator wykonujący wyłuskanie oznaczany jest identycznym symbolem (gwiazdki *) jak operator arytmetyczny mnożenia. Różnica między tymi operatorami, po której zresztą rozpoznaje je kompilator, jest taka, że operator mnożenia jest dwuargumentowy (z jego lewej i prawej strony musi znajdować się wartość lub zmienna), a operator dereferencji – jednoargumentowy.

5 Aby sprawdzić, co się stało po wykonaniu linii 4, ponownie wyświetlamy wartości zmiennych `x` i `px` oraz zawartość komórki pamięci wskazywanej przez wskaźnik `px`. Po kompilacji i uruchomieniu programu na ekranie widzimy. Możemy się więc przekonać, że zmiana zawartości komórki pamięci powoduje również zmianę wartości zmiennej związanej z tą komórką.

```
C:\biblioteczka\program_61\program_61.exe
Wartosc x: 7.456
Wartosc px: 0x22ff74
Wartosc wskazywana przez px: 7.456
Wartosc x: 3.34
Wartosc px: 0x22ff74
Wartosc wskazywana przez px: 3.34
```

Nazwa tablicy jest wskaźnikiem

Tak naprawdę nazwa zmiennej tablicowej jest wskaźnikiem, dzieki któremu możemy poznać adres w pamięci pierwszego elementu tej tablicy. Dlatego możemy nazwę tablicy przypisać bezpośrednio do zmiennej wskaźnikowej, nie wykorzystując do tego celu operatora adresu &.

```
C:\biblioteczka\program_62\program_62.exe
Wartosc wskaznika ptablica1: 0x22ff50
Wartosc wskaznika ptablica2: 0x22ff50
Wartosc dereferencji wskaźnika ptablica2: 2
Wartosc wskaznika (ptablica2+1): 0x22ff54
Wartosc dereferencji wskaźnika (ptablica1+1): 20
Wartosc dereferencji wskaźnika (ptablica1+2): 200
```

W jaki sposób dostać się do kolejnych elementów zmiennej tablicowej? Spójrzmy na następną linię – wyświetlamy w niej wskaźnik `ptablica2` powiększony o 1. Wyświetlony na ekranie adres nie będzie jednak powiększony o 1, ale o 4 bajty. Dlaczego tak się stało? Ponieważ wskaźnik `ptablica2` jest typu `int`, który na przechowanie w pamięci wartości potrzebuje 4 bajtów.

6 Na koniec wyświetlamy dereferencję wskaźnika powiększonego o 1 i 2. Na ekranie zobaczymy wartość drugiego i trzeciego elementu naszej tablicy. Wiemy już więc, w jaki sposób za pomocą wskaźnika odwoływać się do wszystkich elementów tablicy.

Komputer ŚWIAT radzi

Za pomocą wskaźników do tablicy możemy oczywiście bez problemu modyfikować wartości elementów tablicy.

Wskaźnik do łańcucha znaków

Skoro nazwa tablicy jest tak naprawdę wskaźnikiem wskazującym adres pierwszego elementu, w wypadku łańcuchów znakowych można więc, zamiast tablicą, posłużyć się samym wskaźnikiem na tablicę. Kompilator zapisze taki ciąg znaków w pamięci tak, jak tablicę. Sprawdźmy to na przykładzie kodu. Tworzymy w nim wskaźnik, któremu od razu przypisujemy ciąg znaków zapisanych w cudzysłowach. Następnie, wykorzystując nazwę wskaźnika, wyświetlamy ciąg znaków. Po skompilowaniu i uruchomieniu naszego programu na ekranie zostanie wyświetlony napis.

```
char *pnapis = "Komputer SWIAT";
cout << pnapis << endl;
```

Program_63

```
C:\biblioteczka\program_63\program_63.exe
Komputer SWIAT
```

Komputer ŚWIAT radzi

Takiej uproszczonej formy zapisu łańcucha znaków używajmy tylko wtedy, gdy nie zamierzamy w kodzie naszego programu zmieniać treści tego napisu.

Tak więc stworzony przez nas kod jest równoznaczny stworzeniu najpierw tablicy znakowej i przypisaniu jej ciągu znaków, następnie zadeklarowaniu wskaźnika typu znakowego oraz przypisaniu tablicy do tego wskaźnika.

```
char napis[] = "Komputer SWIAT";
char *pnapis;
pnapis = napis;
```

Program_63

Przekazywanie wskaźników do funkcji

Do tej pory argumentami naszych funkcji były zmienne typu liczbowego. Przekazanie do funkcji zmiennej typu wskaźnikowego stwarza całkiem nowe możliwości

Już wiemy, co to są wskaźniki i jak się nimi posługiwać. Dowiedzmy się teraz, co pożytecznego możemy dzięki nim osiągnąć. Komputer ŚWIAT pokaże, w jaki sposób przekazywać wskaźniki jako argumenty do różnych funkcji.

Tablica jako argument funkcji

Często w naszych programach będziemy chcieli do oddzielnej funkcji przekazać kilka lub nawet kilkanaście argumentów. Użycie w takim wypadku zmiennych jest bardzo niewygodne i nieoptymalne (zabiera dużo pamięci i czasu procesora). Najlepszym rozwiązaniem w takim wypadku jest więc przekazanie do funkcji tablicy, w której umieszczamy wszystkie wartości przekazywane do funkcji. Spójrzmy na przykład, którego kod znajdziemy na następnej stronie.

1 Deklarujemy czteroelementową tablicę `liczby` (przechowującą liczby zmiennoprzecinkowe) oraz wskaźnik o nazwie `pliczby`, któremu przypisujemy tablicę `liczby` ❶.

2 Następnie z klawiatury pobieramy cztery liczby ❷ i w pętli wpisujemy je do zmiennej tablicowej `liczby`.

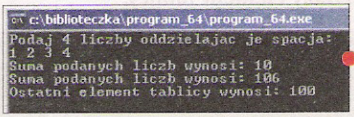
3 W następnych liniach wywołamy funkcję `sumuj()`, więc najpierw zajmijmy się jej napisaniem ❻ (oczywiście nie zapominamy o jej deklaracji, którą umieszczamy na samym początku naszego programu). Zadaniem funkcji będzie zsumowanie czterech liczb przekazanych do niej w postaci tablicy i zwrócenie tej sumy. Dodawanie wykonujemy w pętli for ❼.

4 Wyświetlamy na ekranie wynik działania funkcji `sumuj()`, przekazując do niej nazwę tablicy ❸. Wykonajmy jeszcze raz tę samą linijkę, przekazując tym razem do funkcji wskaźnik `pliczby` wskazujący na naszą tablicę ❹. Na ekra-

► Komputer ŚWIAT radzi

Skoro przekazaliśmy do funkcji adres pierwszego elementu tablicy, to w rzeczywistości podaliśmy funkcji miejsce przechowywania wartości. Przekazanie więc tablicy jako argumentu znacznie się różni od przekazania do funkcji zmiennej - w tym drugim wypadku przekazujemy wyłącznie wartość zmiennej. Sprawdźmy, co się stanie, jeśli wewnątrz funkcji `sumuj()` zmienimy wartość jednego z elementów naszej tablicy (jak pamiętamy, w wypadku zmiennej zmiana wartości wewnątrz funkcji nie miała wpływu na wartość zmiennej poza tą funkcją).

Aby to zrobić, wystarczy usunąć znaki komentarza z linii ❺ oraz ❽ i uruchomić program. Na ekranie zobaczymy widok ●. Widzimy więc, że przekazanie do funkcji tablicy daje możliwość tej funkcji modyfikacji elementów tej tablicy.



```
#include <cstdlib>
#include <iostream>

using namespace std;

float sumuj(float tablica[]);

int main(int argc, char *argv[])
{
    float liczby[4];
    float *pliczby = liczby; 1

    cout << "Podaj 4 liczby oddzielając je spacją: " << endl;
    for (int i=0; i<4; i++) cin >> liczby[i]; 2

    cout << "Suma podanych liczb wynosi: " << sumuj(liczby) << endl; 3
    cout << "Suma podanych liczb wynosi: " << sumuj(pliczby) << endl; 4
//  cout << "Ostatni element tablicy wynosi: " << liczby[3] << endl; 5

    cout << endl << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

float sumuj (float tablica[]) 6
{
      float wynik = 0;
      for (int i=0; i<4; i++) wynik += tablica[i]; 7
//    tablica[3] = 100; 8
      return wynik;
}
```

Program_64

nie zobaczymy widok 1. Co z niego wynika? Między innymi to, że nie ma znaczenia, czy do funkcji przekażemy nazwę tablicy, czy wskaźnik do tej tablicy. Obie zmienne wskazują bowiem na ten sam adres w pamięci, w którym znajduje się pierwszy element tablicy.

```
C:\biblioteczka\program_64\program_64.exe
Podaj 4 liczby oddzielając je spacja:
1 2 3 4
Suma podanych liczb wynosi: 10
Suma podanych liczb wynosi: 10
```

Wskaźnikowe argumenty

Pamiętamy, że przekazanie do funkcji nazwy zmiennej powoduje tak na prawdę przesłanie do tej funkcji kopii wartości zmiennej. W ten sposób nie da się wykonać na przykład funkcji zamieniającej wartości dwóch zmiennych (wartość zmiennej `a` zostaje przypisana do zmiennej `b` i odwrotnie). Jednak taką funkcję możemy bez problemu napisać, korzystając ze wskaźników. Spójrzmy na kod funkcji 1.

1 Definiujemy funkcję o nazwie `zamien1()` niezwracającą żadnego wyniku i jako jej argumenty ustalamy dwa wskaźniki.

```
void zamien1 (int *x, int *y)
{
      int z = *x;
      *x = *y;
      *y = z;
}
```

Program_65

2 Następnie deklarujemy pomocniczą zmienną `z` i przypisujemy jej wartość komórki wskazywanej przez wskaźnik `x`. Teraz wystarczy już pod adres wskazywany przez wskaźnik `x` przypisać wartość z adresu przechowywanego przez zmienną `y`, do którego z kolei przypisujemy wartość zmiennej `z` (znajduje się w niej stara wartość wskazywana przez wskaźnik `x`).

Program_65

```
int a, b;
a = 10;
b = 20;
cout << "a=" << a << "  b=" << b << endl;
zamien1(&a, &b);
cout << "a=" << a << "  b=" << b << endl;
```

3 Aby sprawdzić działanie funkcji `zamien1()`, wewnątrz funkcji `main` deklarujemy dwie zmienne i przypisujemy im dowolne wartości.

4 Następnie wyświetlamy wartości zmiennych i wywołujemy stworzoną przed chwilą funkcję `zamien1()`. Ponieważ argumentami funkcji są wskaźniki (a więc zmienne przechowujące adres), nie możemy przekazać nazw zmiennych, tylko odpowiadające im adresy w pamięci (aby uzyskać adres, wykorzystujemy operator &, który umieszczamy przed nazwą każdej zmiennej).

5 Wyświetlając ponownie wartości zmiennych `a` i `b`, przekonamy się łatwo, że funkcja `zamien1()` działa. Komputer ŚWIAT poleca zapoznanie się z informacjami z ramki Przekazanie przez wskaźnik lub przez referencję. Dowiemy się z niej o jeszcze innym sposobie wykonania takiej funkcji.

```
C:\biblioteczka\program_65\program_65.exe
a=10  b=20
a=20  b=10
```

Przekazanie przez wskaźnik lub przez referencję

Niestety, pamiętanie o tym, aby do danej funkcji przekazywać adresy zmiennych, jest niewygodne. Dlatego przekazywanie wartości przez wskaźnik może być kłopotliwe dla niezbyt zaawansowanych programistów. Oczywiście możemy skorzystać z innego rozwiązania – tak zwanego przekazywania danych przez referencję. Wtedy wywołanie funkcji będzie wyglądało tak (nie musimy korzystać z operatora &). Ze względu na ograniczoną ilość miejsca Komputer ŚWIAT nie będzie tutaj wyjaśniał pojęcia referencji. Warto jednak wiedzieć, że coś takiego istnieje.

```
zamien2(a, b);
```

```
void zamien2 (int &a, int &b)
{
      int x = a;
      a = b;
      b = x;
}
```

Program_65

7 Podstawy programowania obiektowego

Prawie wszystkie napisane przez nas do tej pory programy działają w trybie tekstowym. Sprawdźmy, czy zdobyta przez nas wiedza pozwoli nam na tworzenie okienkowych aplikacji

W rozdziale	strona
Programowanie obiektowe	**113**
Obiekty - właściwości, metody i zdarzenia	113
Tworzymy aplikację	**114**
Dodajemy kod do zdarzeń	117
Ułatwienia C++ Buildera	**120**
Pomoc pod ręką	120
Automatyczne dopisywanie	121
Lista właściwości	121

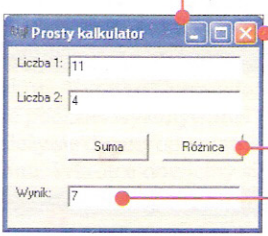
Programowanie liniowe i strukturalne mamy już opanowane. Warto więc chociaż na chwilę porzucić aplikację Dev-C++ i zainstalować bardziej zaawansowane środowisko programistyczne – Borland C++ Builder 6. W tym rozdziale zobaczymy, jak wykorzystać zdobytą przez nas do tej pory wiedzę i stworzyć prawdziwą okienkową aplikację.

Na koniec poznamy również niektóre z bardzo wielu udogodnień środowiska C++ Builder. Dzięki nim stworzymy okno aplikacji z gotowych klocków, które trzeba będzie tylko odpowiednio dopasować do funkcji, które mają pełnić. Oczywiście informacje z tego rozdziału to jedynie przedsmak przygody z programowaniem obiektowym. Wszystkich zainteresowanych tematem Czytelników Komputer ŚWIAT zachęca do pogłębiania zdobytej wiedzy dotyczącej programowania w C++ . Jak się już pewnie przekonaliśmy, programowanie wcale nie jest takie trudne.

Programowanie obiektowe

Wykorzystując obiekty, możemy tworzyć zaawansowane aplikacje. Warto więc wiedzieć, czym one są

Umiemy tworzyć już proste programy działające w oknie wiersza poleceń. Napisaliśmy kilkadziesiąt drobnych kodów – może nie były to zbyt ambitne przedsięwzięcia, ale dzięki nim poznaliśmy podstawy programowania liniowego (kod programu wykonywany jest linijka po linijce) i strukturalnego (kod programu podzielony jest na wykonujące określone czynności funkcje). Na koniec warto poznać chociaż elementarne pojęcia związane z programowaniem obiektowym. Za pomocą obiektów w prosty sposób stworzymy prawdziwą, okienkową aplikację.



Komputer ŚWIAT radzi

Programowanie obiektowe jest złożoną i trudną techniką programowania. Dlatego z tego rozdziału nie dowiemy się wszystkiego na temat obiektów i innych związanych z nimi pojęć. Komputer ŚWIAT na przykładzie prostego programu pokaże jedynie, jak zabrać się do tworzenia nieskomplikowanych aplikacji dla systemu Windows.

Komputer ŚWIAT radzi

Jeśli nie do końca rozumiemy pojęcie obiektu, właściwości czy metody, wyobraźmy sobie jako obiekt na przykład telewizor. Ma on szereg właściwości: wielkość przekątnej ekranu, liczbę programów, które możemy ustawić, aktualnie wyświetlany program czy chociażby kolor obudowy. Telewizor może wykonać też wiele zadań (czyli w języku programowania metod) - można go włączyć i wyłączyć, zmienić kanał czy głośność. Jeśli chcemy na przykład przełączyć program, musimy wydać mu polecenie: włącz kanał X (czyli musimy nacisnąć na odpowiedni przycisk). W programowaniu takie działanie nazywa się wywołaniem metody.

Dla telewizora charakterystyczne są również pewne zdarzenia. Na przykład zdarzenie wywołane przez użytkownika telewizora polegające na włączeniu - jeśli zajdzie takie zdarzenie, uruchomione zostaną podzespoły telewizora i na ekranie pojawi się obraz ustawionego przez nas wcześniej kanału.

Obiekty - właściwości, metody i zdarzenia

W bardzo dużym uproszczeniu można założyć, że obiektem jest każda istniejąca fizycznie w programie wielkość (obiektem jest zmienna, a także okno aplikacji, przycisk czy pole tekstowe). Każdy obiekt ma różne właściwości (własności) oraz metody. Co to są właściwości i metody? Odpowiedź na to pytanie nie jest zbyt trudna – właściwością jest nazwa, kolor, rozmiar, wartość i inne cechy obiektu. Metody to zadania, które dany obiekt potrafi wykonać. Z obiektami związane jest też pojęcie zdarzeń. Zdarzenie jest wystąpieniem określonej sytuacji. Po zajściu takiego zdarzenia program może wykonać odpowiednie, zaprogramowane przez nas czynności.

Tworzymy aplikację

Napiszmy prostą aplikację. Na jej przykładzie dowiemy się, od czego rozpocząć programowanie obiektowe

Pewną dawkę teorii mamy już za sobą. Zobaczmy teraz, w jaki sposób stworzyć okno aplikacji i umieścić na nim różne obiekty (na przykład przyciski). Do tego celu wykorzystamy aplikację Borland C++ Builder 6 Personal. Napiszemy prosty program, który będzie dodawał i odejmował dwie liczby.

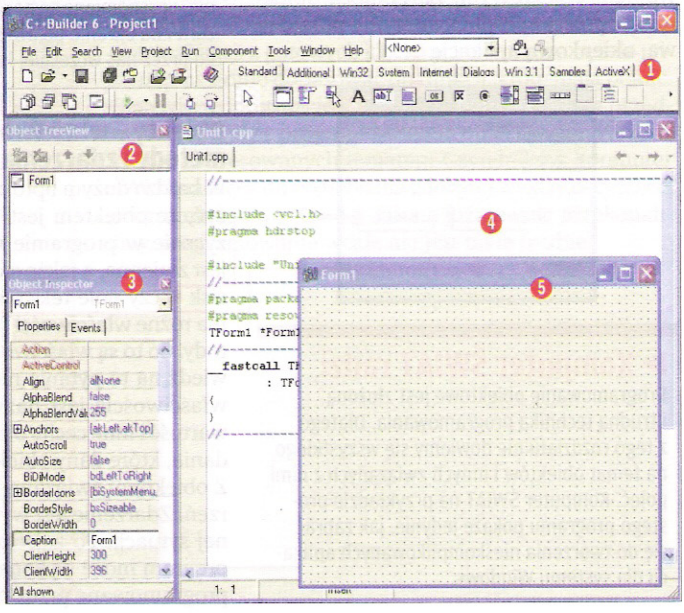
1 Zgodnie z instrukcjami umieszczonymi na płycie dołączonej do tej książki instalujemy środowisko programistyczne Borland C++ Builder 6 Personal. Następnie uruchamiamy zainstalowaną aplikację.

2 Domyślnie, zaraz po otwarciu zostaje stworzony projekt nowego programu. Zapisujemy go od razu na dysku (później warto zapisywać projekt po każdej większej zmianie). W tym celu z menu File wybieramy pozycję Save All. W oknie, które się otwiera, przechodzimy do katalogu C:\biblioteczka\aplikacja (ten ostatni folder musimy stworzyć) i w dwóch kolejnych oknach klikamy na przycisk Zapisz.

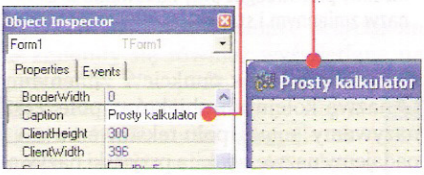
► Komputer ŚWIAT radzi

Borland C++ Builder 6 Personal to, tak jak Dev-C++, zintegrowane środowisko programistyczne, pozwalające na tworzenie programów w języku C++. Aplikacja firmy Borland ma bardzo duże możliwości - oprócz rozbudowanego edytora kodu, zaawansowanego debugera i bardzo dobrej pomocy można w niej na przykład projektować okna aplikacji.

3 Aplikacja składa się z kilku części: okna menu i komponentów ❶, okna obiektów ❷, okna właściwości i zdarzeń ❸, okna kodu ❹ i okna naszego programu ❺ (tak zwanej formatki).



4 Jak już wiemy, okno aplikacji jest obiektem i tak jak każdy obiekt ma pewne właściwości. Jedną z nich jest napis na górnej belce okna. Domyślną wartością tej właściwości jest napis Form1. Zmieńmy więc go. Aby to zrobić, w oknie ❷ klikamy na pozycję Form1 – to nazwa obiektu – okna naszego programu. Następnie w oknie właściwości ❸ odnajdujemy właściwość Caption i w pole obok niej wprowadzamy napis ● i zatwierdzamy klawiszem Enter. Napis pojawia się na górnej belce okna naszego programu ●.



Okno właściwości

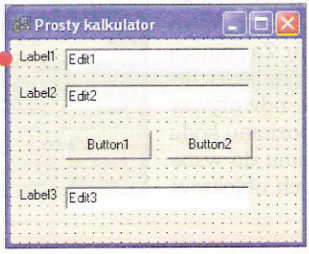
Dzięki oknu właściwości bardzo łatwo można zmieniać wszystkie własności każdego obiektu. W ten sposób jednak ustawimy jedynie początkowe własności obiektu. Aby określoną właściwość zmienić podczas wykonywania programu, należy się do niej odwołać w kodzie programu. Wkrótce dowiemy się, w jaki sposób to robić.

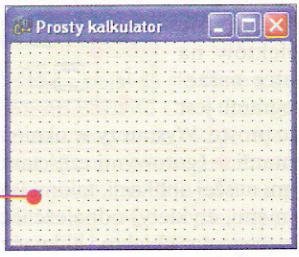
5 Wielu Czytelników jest pewnie ciekawych, co się stanie, jeśli skompilujemy i uruchomimy nasz program. Nic nie stoi na przeszkodzie, aby to zrobić. Z menu Run wybieramy więc pozycję Run – program zostaje skompilowany (kompilacja przebiega w podobny sposób, jak w programie Dev-C++) i uruchomiony. Na ekranie pojawia się okno naszej aplikacji ●. Chyba jedyne, co możemy z nim w tej chwili zrobić, to zamknąć je i zająć się rozbudową naszego dzieła.

Komputer ŚWIAT radzi

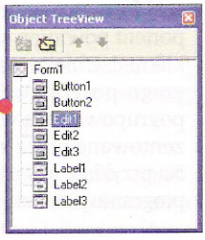
Program możemy również uruchamiać tak jak w aplikacji Dev-C++ - wciskając klawisz F9.

6 W oknie naszego programu umieśćmy teraz inne obiekty: trzy pola tekstowe, dwa przyciski i trzy etykiety ●. O tym, jak to zrobić, dowiemy się z zamieszczonej na następnej stronie ramki Dodajemy komponenty.

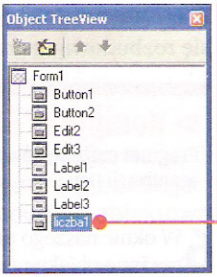


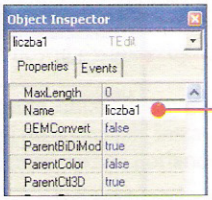


7 Domyślne nazwy wszystkich dodanych przez nas komponentów pojawiły się w oknie ●. Jak widzimy, wszystkie dodane obiekty zostały umieszczone wewnątrz obiektu



naszego okna (które nazywa się Form1 – możemy to sprawdzić w oknie właściwości po kliknięciu na belkę tytułową naszego programu). Tak więc wszystkie dodane przez nas obiekty zawierają się w obiekcie Form1 (są podrzędne w stosunku do niego). Nazwa każdego z obiektów składa się z nazwy danego komponentu i kolejnego numeru. Za pomocą tych nazw będziemy mogli odwoływać się z poziomu kodu programu do wstawionych obiektów.





Warto więc je zmienić na bardziej przystępne. Aby to zrobić, klikamy na dany obiekt (na przykład pole Edit1) i w oknie właściwości odszukujemy własność Name. W pole obok wpisujemy nową nazwę – na przykład . Po zatwierdzeniu klawiszem enter nazwa obiektu zmienia się .

Komputer ŚWIAT radzi

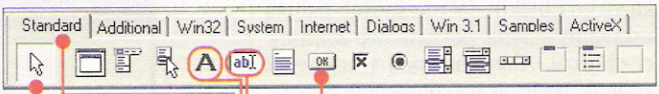
Oczywiście nazwy obiektów muszą spełniać określone kryteria (podobne do tych, których musimy przestrzegać przy nadawaniu w C++ nazw zmiennym i stałym).

8 Podobnie jak w punkcie 7 zmieniamy nazwę kolejnych obiektów: pole Edit2 nazywamy liczba2, polu tekstowemu Edit3 nadajemy nazwę wynik, a przyciski nazywamy kolejno suma i roznica. Nazwy etykiet możemy pozostawić bez zmian.

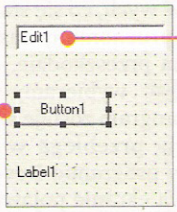
Dodajemy komponenty

Umieszczanie obiektów na formatce naszego programu jest bardzo proste. Wystarczy w oknie z komponentami (gotowymi częściami składowymi naszej aplikacji, takimi jak przyciski, menu, etykiety) odnaleźć ikonę interesującego nas obiektu i kliknąć na nią. Następnie należy ustawić kursor na naszej formatce w miejscu, w którym komponent powinien się pojawić, i ponownie kliknąć. Obiekt zostaje wstawiony do naszego programu. Uwaga! Komponenty pogrupowane są w różne kategorie reprezentowane przez zakładki .

Na przykład, aby wstawić do okna naszego programu przycisk, klikamy na ikonę i następnie klikamy w dowolnym miejscu naszej formatki. Przycisk pojawia się na niej. W identyczny sposób możemy wstawić inne komponenty, na przykład pole tekstowe (reprezentowane przez ikonę) czy etykietę (którą symbolizuje ikona).

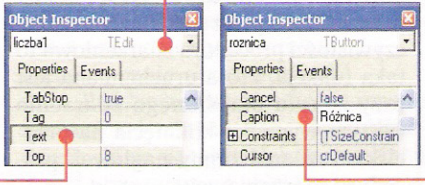


Bardzo łatwo możemy zmienić rozmiar wstawionych elementów – wystarczy kliknąć na dany komponent i za pomocą kwadratów rozciągnąć go lub zmniejszyć.

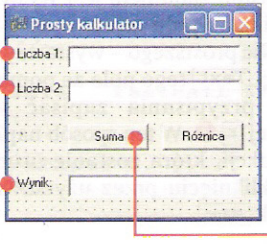


► Komputer ŚWIAT radzi

Wygodnym sposobem oznaczenia obiektu, którego właściwości chcemy zmienić, jest wybranie go z listy.



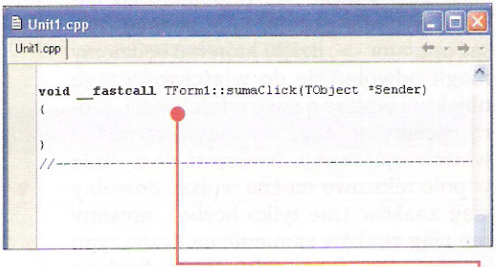
9 Po nadaniu nazw polom i przyciskom zmieniły się również wyświetlane na nich napisy. W wypadku pól napisy te nie są nam w ogóle potrzebne. Aby je skasować, klikamy na dane pole tekstowe, w oknie właściwości odnajdujemy własność i usuwamy tekst z pola obok tej własności. Podobnie robimy z pozostałymi polami tekstowymi. W wypadku przycisków odnajdujemy właściwości i modyfikujemy znajdujące się w nich wartości (dodajemy wielkie litery i polskie znaki). Na koniec modyfikujemy jeszcze właściwości Caption, tym razem dla etykiet tekstowych. Okno naszego programu wygląda już całkiem nieźle. Możemy skompilować i uruchomić program – niestety klikanie na przyciski nie powoduje żadnych akcji. Zajmijmy się więc oprogramowaniem przycisku Suma i Różnica.



Dodajemy kod do zdarzeń

Interfejs naszego programu jest już gotowy. Wystarczy tylko teraz dopisać kod, który pobierze wpisane przez użytkownika wartości z pól o nazwach liczba1 oraz liczba2 i w zależności od tego, na który przycisk użytkownik kliknie, w polu wynik wyświetli wynik dodawania lub odejmowania tych liczb.

1 W jaki sposób do programu dodać kod, który zostanie wykonany po kliknięciu przez użytkownika na przycisk Suma? Wystarczy w oknie projektu naszej aplikacji dwukrotnie kliknąć na ten przycisk.



2 Na ekranie pojawia się okno z kodem, w którym Borland C++ Builder wygenerował automatycznie kod procedury zdarzeniowej. Zostanie ona uruchomiona po kliknięciu przez użytkownika na przycisk suma. Możemy więc w niej wpisać kod, który doda do siebie liczby z dwóch pól tekstowych i wynik tej operacji wyświetli w trzecim polu.

3 Na początku tworzymy trzy zmienne typu `float` (patrz 1 w kodzie na następnej stronie).

4 W zmiennej x musimy teraz zapisać wartość z pierwszego pola tekstowego. W jaki sposób to zrobić? Bardzo łatwo.

Unit1.cpp

```
void __fastcall TForm1::sumaClick(TObject *Sender)
{
  float x, y, z; 1
  x = StrToFloat(liczba1->Text); 2
  y = StrToFloat(liczba2->Text); 3
  z = x + y; 4
  wynik->Text = z; 5
}
//---------------------------------------------------
```

Wystarczy wpisać nazwę obiektu (w naszym wypadku pola liczba1), później wpisać operator ->, dzięki któremu będziemy mogli odwołać się do właściwości tego obiektu i wpisać nazwę właściwości, którą chcemy pobrać (w naszym wypadku właściwość `Text`). Ponieważ domyślnie w pole tekstowe można wpisać dowolny ciąg znaków (nie tylko liczbę), musimy ten ciąg znaków zamienić na liczbę typu `float`. Do tego celu służy funkcja `StrToFloat` 2.

5 Identycznie postępujemy ze zmienną `y`, do której przypisujemy liczbę z pola tekstowego liczba2 3.

6 Mając dwie liczby, możemy je do siebie dodać i wynik tej operacji zapisać w zmiennej `z` 4.

▶ Komputer ŚWIAT radzi

Oczywiście w kodzie naszego programu możemy korzystać z wszystkich poznanych przez nas w poprzednich rozdziałach tej książki konstrukcji języka C++ – instrukcji warunkowych, pętli czy tablic.

▼ Wygląd procedury zdarzeniowej

Nagłówek wygenerowanej przez aplikację Borland C++ Builder 6 procedury zdarzeniowej przypomina nagłówek funkcji, które tworzyliśmy w Dev-C++ (patrz rozdział 5). W rzeczywistości procedura zdarzeniowa to taka funkcja, która uruchamiana jest przy zajściu określonego zdarzenia (w tym wypadku kliknięcia na przycisk). Przeanalizujmy nagłówek procedury zdarzeniowej:

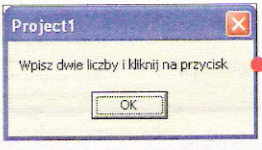
- słowo kluczowe `void` oznacza, że funkcja nie będzie zwracała żadnej wartości,
- słowo `fastcall` (poprzedzone dwoma znakami podkreślenia) określa sposób przekazywania argumentów funkcji – `fastcall` dodawany jest do każdej funkcji, więc nie musimy się nim zbytnio przejmować,
- `TForm1::sumaClick` to nazwa naszej funkcji – została ona stworzona między innymi z nazwy przycisku (na przykład suma) oraz nazwy zdarzenia, które spowoduje jej uruchomienie (`Click` – kliknięcie).
- w nawiasach okrągłych znajduje się argument przekazywany do funkcji. Jest nim obiekt, który wywołał tę funkcję.

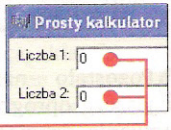
7 Ostatnim zadaniem jest wyświetlenie wyniku w polu tekstowym o nazwie wynik. Nic prostszego – wystarczy do właściwości `Text` tego pola za pomocą operatora przypisania zapisać wartość zmiennej `z` 5. W ten sposób napisaliśmy kod funkcji, która zostanie uruchomiona po kliknięciu przez użytkownika na przycisk suma.

Podglądamy kod

Obie stworzone przez nas funkcje znajdują się w tym samym pliku źródłowym. Oprócz nich, tak jak w programach pisanych przez nas w Dev-C++, znajdziemy stworzone automatycznie przez Borland C++ Builder instrukcje (między innymi znana nam dyrektywa `#include`). W kodzie znajduje się także globalna zmienna `Form1`, wskazująca na obiekt okna naszego programu, oraz szkielet funkcji. Możemy do niej wpisać instrukcje, które zostaną wykonane podczas uruchamiania naszego programu. Na przykład dopisanie instrukcji spowoduje wyświetlenie na ekranie komunikatu, a instrukcji – wpisanie w pola tekstowe wartości początkowych.

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
 ShowMessage("Wpisz dwie liczby i kliknij na przycisk");

 liczba1->Text = 0;
 liczba2->Text = 0;
}
```



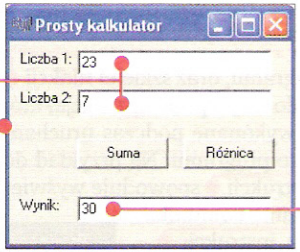


Unit1.cpp

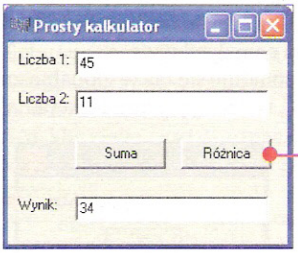
Unit1.cpp

```
//---------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------

void __fastcall TForm1::sumaClick(TObject *Sender)
{

  float x, y, z;
```

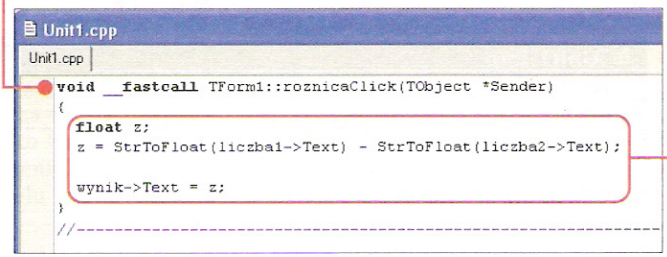
8 Możemy teraz skompilować i uruchomić nasz program (wciskając klawisz F9). Na ekranie pojawia się okno. Po wpisaniu dwóch liczb i kliknięciu na przycisk suma w polu widzimy poprawny wynik.



generowaną funkcją. Wewnątrz niej wpisujemy kod. Jego działanie jest takie samo, jak w wypadku poprzedniej funkcji. Oczywiście tym razem wykonywane jest odejmowanie zamiast dodawania, a kod został znacznie uproszczony. Po uruchomieniu naszego programu możemy przetestować działanie przycisku roznica.



9 Pozostało nam jeszcze oprogramowanie drugiego przycisku. Kończymy więc działanie naszego programu i dwukrotnie klikamy na roznica. Pojawia się okno z kodem naszego programu oraz z automatycznie wy-



Ułatwienia C++ Buildera

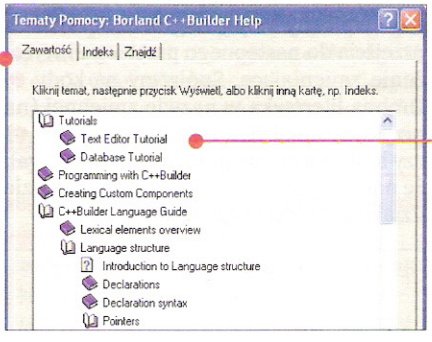
Aplikacja firmy Borland ma wiele ciekawych możliwości upraszczających programowanie. Warto je poznać

Dzięki wielu ułatwieniom oferowanym przez aplikację Borland C++ Builder programowanie nie polega już na wstukiwaniu niezrozumiałych poleceń, przypomina raczej budowanie z klocków, które w środowisku programistycznym otrzymujemy niemal jak na tacy. Warto więc poznać niektóre możliwości C++ Buildera wspomagające programowanie – dzięki nim pisanie programów będzie znacznie przyjemniejsze.

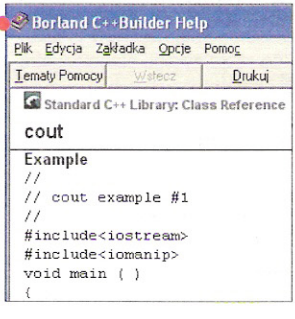
Pomoc pod ręką

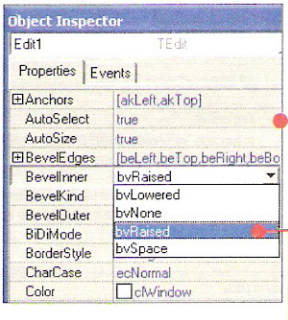
C++ Builder wyposażony został w bardzo obszerną pomoc dotyczącą nie tylko obsługi środowiska, ale również języka C++. Zobaczmy, jak z niej korzystać.

1 Z menu Help wybieramy pozycję C++Builder Help. Otwiera się okno z tematami pomocy. Znajdziemy w nim na przykład poradnik, w jaki sposób krok po kroku stworzyć edytor tekstowy w C++, a także wyjaśnienia różnych konstrukcji językowych. Wystarczy dwukrotnie kliknąć na pozycję z ikoną ?.



2 W każdej chwili możemy także wywołać tak zwaną pomoc kontekstową (związaną ze słowem kluczowym, wewnątrz którego znajduje się kursor). W tym celu wystarczy wcisnąć klawisz F1 – jeśli z danym słowem wiąże się jakiś temat pomocy, zostanie wyświetlone okno, w którym często oprócz opisu danej instrukcji, znajdziemy przykład jej zastosowania.



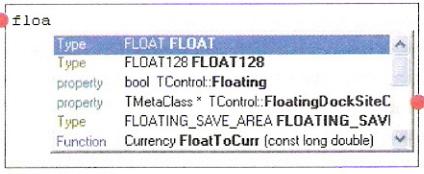


► Komputer ŚWIAT radzi

Niestety, pomoc w aplikacji Borland C++ Builder jest w języku angielskim. Jeśli nawet nie posługujemy się biegle tym językiem, to warto korzystać z pomocy. Znajdziemy w niej wiele konkretnych przykładowych kodów przedstawiających praktyczne zastosowanie danej konstrukcji języka C++.

Automatyczne dopisywanie

Dzięki automatycznemu dokańczaniu poleceń przez środowisko C++ Builder programowanie może być znacznie szybsze. Na czym polega to udogodnienie? Wpiszmy na przykład początek nazwy typu float i wciśnijmy kombinację klawiszy Ctrl spacja. Jeśli wpisany przez nas początek odpowiada nazwie tylko jednego polecenia, zostaje ono automatycznie wpisane. Jeśli jest więcej instrukcji rozpoczynających się od wpisanej przez nas frazy, zostaje wyświetlone okno, z którego możemy wybrać odpowiadające nam polecenie.



Lista właściwości

Każdy obiekt ma określone właściwości (nazwę, kolor czy wielkość), które możemy edytować w oknie. Wielu z nich nie musimy jednak wpisywać ręcznie, środowisko C++ Builder umożliwia nam wybór odpowiedniej właściwości z listy (na przykład).

Błędy w kodzie

Programowanie to czasochłonne zajęcie. Warto więc wiedzieć, na co zwrócić uwagę, aby napisany przez nas program dał się skompilować i był wolny od błędów

Nawet najlepszy programista nie jest w stanie napisać każdego programu bezbłędnie. Niekiedy pomyłka będzie konsekwencją braku średnika na końcu linii, innym razem będzie to literówka w nazwie zmiennej. Takie błędy są jednak niegroźne – program się nie skompiluje, a kompilator wyświetli listę nieprawidłowości wraz z prawdopodobną przyczyną błędu (na przykład). W tabeli na następnych stronach Komputer ŚWIAT wymienia najczęściej występujące komunikaty błędów i podpowiada, jak najszybciej pozbyć się nieprawidłowości.

Groźne pomyłki

Błędy z tabeli, która znajduje się na następnych stronach książki, są łatwe do poprawienia. Znacznie trudniej odnaleźć w kodzie błąd, który z punktu widzenia kompilatora jest poprawnym zapisem, jednak powoduje, że nasz program nie działa prawidłowo (co czasem jest niezwykle trudno wykryć, bo na przykład objawia się tylko w kilku szczególnych sytuacjach). Poznajmy niektóre miejsca w kodzie, na które warto zwrócić szczególną uwagę podczas kodowania oraz wtedy, gdy nasz program nie działa tak, jak zakładaliśmy.

Problemy z pętlami

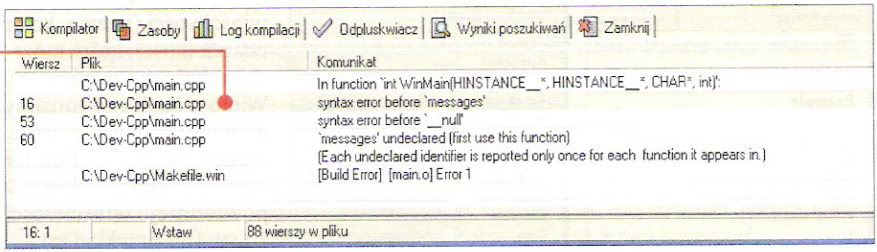
Pętla jest często wykorzystywaną konstrukcją języka. Dlatego warto kilka razy sprawdzić wpisany w jej nagłówku warunek przejścia do następnego przebiegu i wyrażenie zmieniające. Spójrzmy na kody . Drobna literówka w nazwie zmiennej (na przykład brak cyfry 1 na końcu nazwy) czy brak znaku mniejszości spowodował, że nasza pętla zamiast 10 wykonań będzie działała w nieskończoność.

```
for (int liczba1 = 0; liczba <= 9; liczba1++)
{
    cout << liczba1;
}
```

```
for (int i = 0; i <= 9; i++)
{
    cout << "Napis";
}
```

Obliczenia nie po kolei

Na stronie 39 mogliśmy przeczytać, że w wyrażeniach z wykorzystaniem operatorów arytmetycznych (+, -, /, *) obowiązują podobne zasady, jak w matematyce – najpierw wykonywane jest mnożenie i dzielenie, a dopiero później doda-



wanie i odejmowanie. Warto również wiedzieć, że dzielenie modulo, oznaczane symbolem %, jest równorzędne mnożeniu i dzieleniu. Na samym końcu wykonywana jest operacja przypisania. W tabeli Komputer ŚWIAT przedstawił priorytety kolejnych operatorów (od najwyższego do najniższego). Operacje z największym priorytetem wykonywane są na samym początku. Operatory mające ten sam priorytet wykonywane są kolejno.

Priorytet operacji

priorytet	operator	działanie
największy	*, /, %	mnożenie, dzielenie, dzielenie modulo
średni	+, -	dodawanie, odejmowanie
najmniejszy	=	przypisanie

Komputer ŚWIAT radzi

Sytuacja z kolejnością wykonywania operacji komplikuje się w wypadku stosowania operacji inkrementacji i dekrementacji. Operatory ++ i -- mają największy priorytet i wykonywane są na samym początku. Należy jednak pamiętać, że występują one w dwóch wersjach - przedrostkowej i przyrostkowej. Inkrementacja i dekrementacja przedrostkowa wykonywana jest przed wykorzystaniem zmiennej, a przyrostkowa dopiero po wykorzystaniu zmiennej. Tak więc wyrażenie `y = 3 * (++x)++ - 2;` (przy założeniu, że zmienne `x` i `y` mają na początku zerową wartość) zostanie wykonane w taki sposób: przeprowadzona zostanie inkrementacja `++x`, następnie mnożenie, później odejmowanie i przypisanie obliczonej wartości wyrażenia do zmiennej `y`. Dopiero po tym przypisaniu zostanie wykonana inkrementacja `x++`. Tak więc w konsekwencji `x` przyjmie wartość 2, a `y` wartość 1. Jak więc widzimy, w tego typu wyrażeniach może czaić się szereg pomyłek, które są niezwykle trudne do zidentyfikowania, a powodują błędne działanie naszego programu.

Oczywiście kolejność operacji możemy niemal dowolnie zmienić za pomocą nawiasów okrągłych.

W wypadku nieprawidłowości w działaniu naszego programu warto więc przyjrzeć się bardziej skomplikowanym wyrażeniom i dokładnie je przeanalizować.

Błędny wynik obliczeń

Jeśli zadaniem naszego programu jest wykonanie bardziej lub mniej skomplikowanych obliczeń, dobrze jest przeanalizować znajdujące się w kodzie wyrażenia. Warto zwrócić szczególną uwagę na typy występujących wewnątrz nich zmiennych. Spójrzmy na przykład na kod . Założeniem programisty było przypisane do zmiennej `wynik` wyniku obliczeń 4*x/y. Jednak najpierw zostało obliczone wyrażenie pośrednie x/y, którego wynik zapisano w zmiennej całkowitoliczbowej `z` (czyli wynik został pozbawiony części ułamkowej) i wykorzystano później do obliczenia wyniku końcowego. W efekcie do zmiennej `wynik` zostanie zapisana nieprawidłowa liczba 20. Gdybyśmy natomiast skorzystali z wyrażenia `wynik = 4 * x/y;`, byłaby to liczba 22.

```
int x, y, z;
float wynik;
x = 11;
y = 2;
z = x / y;

wynik = 4 * z;
```

Co znaczą komunikaty o błędach w Dev-C++

Treść komunikatu błędu lub ostrzeżenia	Przyczyna powstania błędu lub ostrzeżenia
'XXX' undeclared (first use this function)	Zmienna lub funkcja XXX nie została wcześniej zadeklarowana
[Warning] no newline at end of file	Brak pustej linijki na końcu pliku
[Warning] assignment to 'int' from 'double'	Przypisanie do zmiennej całkowitoliczbowej liczby zmiennoprzecinkowej
Invalid conversion from 'const char*' to 'char'	Przypisanie do zmiennej typu znakowego (char) napisu lub znaku umieszczonego w cudzysłowach
Syntax error before 'XXX'	Błąd składni
Syntax error at end of input	Brak zamykającego nawiasu klamrowego
'main' must return 'int'	Źle zadeklarowana funkcja główna
[Warning] division by zero in 'XXX'	Dzielenie przez zero
Missing terminating > character	Brak nawiasu ostrego zamykającego
Too few arguments to function 'XXX' at this point in file	Przy wywołaniu funkcji XXX podano za mało argumentów
Too many arguments to function 'XXX' at this point in file	Przy wywołaniu funkcji XXX podano za dużo argumentów
Storage size of 'XXX' isn't known	Niezadeklarowany rozmiar tablic

Sposoby wyeliminowania błędu i ostrzeżenia

1. Jeśli XXX to nazwa używanej zmiennej, komunikat oznacza, że nie została ona wcześniej zadeklarowana (patrz strona 36)
2. Jeśli XXX to nazwa napisanej przez nas funkcji, należy deklarację tej funkcji umieścić przed jej pierwszym wywołaniem (patrz strona 85)
3. Jeśli XXX dotyczy funkcji standardowej języka C++, należy za pomocą dyrektywy `#include` dodać na początku kodu plik z definicją tej funkcji (na przykład jeśli komunikat dotyczy funkcji `cout`, na początku kodu należy umieścić instrukcję `#include <iostream>`) (patrz strona 26)
4. Jeśli XXX dotyczy funkcji standardowej, a plik z jej definicją został dołączony, należy dopisać instrukcję `using namespace std;` (patrz strona 26)

Należy na końcu kodu źródłowego umieścić pustą linię

Należy zmienić typ zmiennej na zmiennoprzecinkowy lub zamienić przypisywaną liczbę lub zmienną na całkowitą (patrz ramka na stronie 81)

1. Jeśli do zmiennej znakowej przypisujemy pojedynczy znak, należy zamienić cudzysłowy na apostrofy
2. Jeśli chcemy do zmiennej wpisać napis, musimy zmienić typ zmiennej na łańcuch znakowy (patrz strona 102) lub typ `string` (patrz strona 103)

Najbardziej prawdopodobną przyczyną błędu jest brak średnika w linii poprzedzającej tę, w której wystąpił błąd (patrz strona 24) lub błąd w składni instrukcji (na przykład przecinek zamiast średnika pomiędzy wyrażeniami w nagłówku pętli)

Należy sprawdzić, czy każdy nawias klamrowy { ma w kodzie swój odpowiednik } zamykający blok instrukcji

Trzeba zmienić typ funkcji głównej `main` na typ całkowitoliczbowy (patrz strona 27)

Należy tak zmodyfikować kod, aby nie występowało dzielenie przez 0

Należy dopisać brakujący znak >

Należy podać tyle argumentów funkcji, ile dla niej zadeklarowaliśmy

Należy podać tyle argumentów funkcji, ile dla niej zadeklarowaliśmy

Należy wewnątrz nawiasów kwadratowych podczas deklaracji tablicy wpisać rozmiar tablicy lub zainicjować ją wartościami (patrz strona 97)

Asembler - język programowania niskiego poziomu. Polecenia Asemblera odpowiadają rozkazom procesora i mają postać symbolicznych oznaczeń literowych (na przykład mov ds, ax). Asembler jest bardzo skomplikowanym językiem i wymaga od programisty bardzo dużej wiedzy technicznej.

```
mov ax, $b625
mov es, ax
mov al, 24
mov ah, 0
mov bx, 0
mov es:[bx], al
inc bx
mov es:[bx], ah
inc bx
cmp bx, 2000
```

char - oznaczenie typu danych. Zmienna typu char (inaczej zwana zmienną znakową) może przechowywać jeden znak (a ściślej mówiąc, liczbę całkowitą reprezentującą dany znak, na przykład literę T).

debuger - zwane również odpluskwiaczem specjalne narzędzie środowiska programistycznego, które umożliwia dokładne kontrolowanie uruchamianego programu. Debuger ułatwia wyszukiwanie błędów w kodzie źródłowym, pozwala na wykonywanie programu krok po kroku i kontrolowanie na bieżąco wartości zmiennych.

float - oznaczenie typu danych. Zmienna typu float może przechowywać liczbę zmiennopozycyjną (zwaną również zmiennoprzecinkową), czyli liczbę składającą się z części całkowitej i ułamkowej.

funkcja - wydzielony fragment kodu programu służący najczęściej do wykonywania określonego zadania (na przykład obliczania pola koła). Do funkcji można przekazać jeden lub więcej argumentów (na przykład długość promienia). Funkcja może też zwracać wynik swojego działania (na przykład obliczone wewnątrz niej pole koła).

instrukcja - konstrukcja językowa oznaczająca wykonanie pewnej czynności - przypisania wartości do zmiennej czy wyświetlenia napisu na ekranie.

instrukcja warunkowa - konstrukcja języka pozwalająca na wykonanie różnych części kodu źródłowego w zależności od spełnienia określonego warunku.

int - oznaczenie typu danych. Zmienna typu int przechowuje jedną liczbę całkowitą.

język niskiego poziomu - język programowania, którego składnia jest bardzo zbliżona do kodu maszynowego i niezrozumiała dla niewtajemniczonej osoby. Przykładem języka programowania niskiego poziomu jest Asembler.

język wysokiego poziomu - język programowania pozwalający na tworzenie aplikacji za pomocą konstrukcji bardziej zbliżonych do języka ludzkiego niż do kodu maszynowego zrozumiałego dla komputerów, na przykład C++.

kod maszynowy - język zrozumiały dla procesora. Kod maszynowy składa się z ciągu zer i jedynek, które stanowią rozkazy dla procesora.

```
01010100011101110011100011111001101
11100111000111111011010101011110000
11100011100111000111110011000111100
```

kompilacja - proces tłumaczenia kodu źródłowego zapisanego w języku wysokiego poziomu na kod maszynowy zrozumiały dla procesora. Podczas kompilacji odbywa się często również linkowanie (dołączanie) bibliotek i plików niezbędnych do uruchomienia skompilowanego programu.

kompilator - program, najczęściej część środowiska programistycznego, wykonujący kompilację (czyli tłumaczenie kodu źródłowego na język maszynowy).

linker - zwany również konsolidatorem program łączący z napisanym przez nas kodem wszystkie niezbędne pliki i biblioteki pozwalające na uruchomienie i działanie skompilowanego programu.

metoda obiektu - zadanie, które może wykonywać obiekt. Każdy obiekt ma zestaw charakterystycznych dla niego metod. Na przykład me-

todami pola tekstowego (obiektu) mogą być: wyczyszczenie zawartości, skopiowanie treści do schowka czy zaznaczenie znajdującego się w polu napisu.

obiekt - moduł (fragment kodu) odpowiadający za wykonywanie określonej czynności. Obiekt ma pewne właściwości (na przykład kolor, wielkość, nazwę) oraz metody (zadania, które obiekt potrafi wykonać). Obiektem jest na przykład zmienna, pole tekstowe, przycisk czy okno aplikacji.

odpluskwiacz - patrz debuger.

pętla - konstrukcja języka programowania, pozwalająca na wykonywanie określonych czynności wiele razy. Pętle mogą wykonywać instrukcje w nieskończoność lub do momentu, w którym zostanie spełniony ustalony przez nas warunek.

stała - wartość przypisana do pewnej nazwy i ustawiona w kodzie na stałe. Na przykład do stałej pi możemy przypisać wartość 3,1415 i później z niej korzystać w kodzie.

string - oznaczenie typu danych. Zmienna typu string może przechowywać napisy.

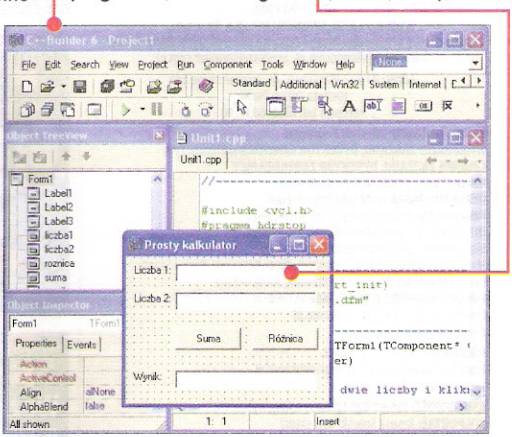
tablica - zmienna pozwalająca na przechowywanie w jednym miejscu kilku lub kilkunastu różnych wartości tego samego typu. Do każdej wartości możemy odwołać się przez nazwę tablicy i indeks, czyli liczbę określającą daną wartość w tablicy.

typ danych zmiennej - określa rodzaj danych, które mogą być przechowywane w zmiennej. Mogą to być na przykład liczby całkowite, zmiennopozycyjne czy znaki.

właściwość obiektu - cecha obiektu, którą można odczytać i w większości wypadków również zmienić. Przykładowymi właściwościami pola tekstowego (obiektu) są: nazwa pola, tekst znajdujący się wewnątrz niego, rozmiar czy rodzaj czcionki.

wskaźnik - zmienna typu wskaźnikowego, która przechowuje adres komórki pamięci, w której najczęściej znajduje się wartość innej zmiennej.

Zintegrowane Środowisko Programistyczne (IDE - ang. Integrated Development Environment) - aplikacja lub kilka aplikacji służących do tworzenia, modyfikowania i testowania programów. W aplikacjach IDE można edytować i kompilować kod źródłowy lub tworzyć zasoby programu (okna dialogowe, menu, ikony).



zdarzenie - zajście określonej sytuacji (na przykład kliknięcie lewym przyciskiem myszy na dany przycisk). Program może po zajściu takiego zdarzenia wykonać zaprogramowane czynności (na przykład wyświetlić nowe okno).

zmienna - nazwany obszar pamięci, w którym przechowywana jest wartość określonego typu. Każda zmienna może przechowywać określony typ informacji. Na przykład w zmiennej typu int można zapisać jedną liczbę całkowitą.

Stopka redakcyjna

Extra Komputer Świat

od 1999 roku

Adres redakcji:
02-337 Warszawa, ul. Mszczonowska 2

Kontakt z Czytelnikami:
tel. (022) 4587606 (w godzinach 10-17),
www.komputerswiat.pl, biblioteczka@komputerswiat.pl

DYREKTOR WYDAWNICZY: Marcin Przasnyski, Joanna Chudziak (z-ca)
REDAKTOR NACZELNY: Wiesław Małecki
SEKRETARZ REDAKCJI: Ewelina Markuszewska, Agnieszka Al-Jawahiri
AUTOR: Paweł Paczuski
FOTO: Krystyna Pagowska
DZIAŁ DTP: Katarzyna Ochelska, Izabela Królikowska, Wojciech Paszkowski, Sebastian Sękalski
KOREKTA: Jolanta Rososińska

axel springer

od 1994 roku
Wydawca: AXEL SPRINGER POLSKA Sp. z o.o.,
Członek Izby Wydawców Prasy i Związku Kontroli Dystrybucji Prasy
www.axelspringer.com.pl, asp@axelspringer.com.pl

Adres: 02-222 Warszawa, Al. Jerozolimskie 181
(Ochota Office Park),
recepcja tel. (022) 6084100

PREZES ZARZĄDU: Wiesław Podkański
WICEPREZES ZARZĄDU, DYREKTOR GENERALNY: Florian Fels
CZŁONEK ZARZĄDU, DYREKTOR ZARZĄDZAJĄCY MARKETING i SPRZEDAŻ: Beata Mońka
DYREKTOR BIURA ZARZĄDU: Michał Fijoł
FINANSE: Paweł Doktór (dyrektor finansowy)
REKLAMA: tel. (022) 6084300, (022) 6084118,
biblioteczka.reklama@axelspringer.com.pl
Małgorzata Cetera-Bulka (dyrektor), Piotr Roszczyk, Paweł Stano, Karol Karwowski, Magdalena Jóźwiak, Anna Grecka
PUBLIC RELATIONS & EVENT MARKETING: tel. (022) 6084178,
biblioteczka.pr@axelspringer.com.pl
Aneta Pacuszka

PROMOCJA: tel. (022) 6084263,
biblioteczka.promocja@axelspringer.com.pl
Jacek Wlazło (dyrektor)

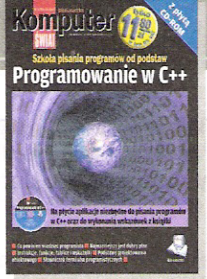
KSIĘGOWOŚĆ: Janusz Bąk (główny księgowy)
KOLPORTAŻ: Janusz Snarski (dyrektor)
PRODUKCJA: Mariusz Gajda (dyrektor)
DRUK: DRUK-INTRO S.A., Inowrocław
EGZEMPLARZE ARCHIWALNE: tel. (022) 6084002, faks (022) 6084007
infolinia tel. 0801 120003 w godzinach 9-17, prenumerata@axel-springer.com.pl

PRENUMERATA KRAJOWA: Poczta Polska oraz „Ruch" S.A. na terenie całego kraju. Prenumerata zagraniczna: RUCH S.A. tel. (022) 5328819 www.ruch.com.pl lub www.exportim.com

Redakcja nie zwraca materiałów niezamówionych, zastrzega sobie prawo redagowania nadesłanych tekstów, nie odpowiada za treść zamieszczonych reklam i ogłoszeń.

NAKŁAD KONTROLOWANY
ZWIĄZEK KONTROLI DYSTRYBUCJI PRASY

Zabroniona jest bezumowna sprzedaż czasopisma po cenie niższej od ceny detalicznej ustalonej przez wydawcę. Sprzedaż numerów aktualnych i archiwalnych po innej cenie jest nielegalna i grozi odpowiedzialnością karną.



Teraz kupisz w Empikach w całym kraju

Biblioteczka

Komputer ŚWIAT

Pisanie programów to żadna sztuka

Programowanie to umiejętność tworzenia aplikacji. Aby powstał ciekawy i użyteczny program, nie wystarczy sam pomysł. Niezbędna jest również wiedza na temat języka programowania, za pomocą którego mamy zamiar stworzyć nasze dzieło. Oddając się lekturze najnowszej książki Komputer ŚWIATA, szybko poznamy najważniejsze reguły języka C++, w którym będziemy tworzyć programy. Dzięki wielu ciekawym przykładom bez problemu zrozumiemy działanie z pozoru trudnych i skomplikowanych konstrukcji językowych. Oczywiście przeczytanie tej książki, nawet od deski do deski, nie sprawi, że zaraz napiszemy nowy system operacyjny. Zdobędziemy jednak praktyczną wiedzę, która wystarczy nam do pisania prostych aplikacji i będzie doskonałym punktem wyjścia do dalszej nauki programowania.

ISBN 83-89371-33-2
9 788389 371331 01>

CENA 11,80 zł

Biblioteczka